DOCUMENT RESUME

ED 078 647 EM 011 177

AUTHOR Blount, Sumner E.

TITLE A Generative CAI Monitor For Teaching Machine-Language Programming.

INSTITUTION Connecticut Univ., Storrs. Dept. of Electrical Engineering.

SPONS AGENCY Connecticut Research Commission, Hartford.; National Center for Educational Research and Development (DHEW/OE), Washington, D.C.
PUB DATE May 72
GRANT OEG-0-72-0895
NOTE 132p.

EDRS PRICE MF-$0.65 HC-$6.58
DESCRIPTORS *Computer Assisted Instruction; *Computers; Feedback; Individualized Instruction; Problem Solving; Program Descriptions; *Programing; *Programing Languages
IDENTIFIERS IBM 360; *Machine Language Teaching; MALT; SEDCOM; Simulated Educational Computer

ABSTRACT

An effective generative computer-assisted instructional system designed to teach basic machine language computer programing is now available. The system--known as Machine Language Teacher (MALT)--is implemented on an IBM 360 with numerous 2741 remote terminals giving student access. It teaches the machine language of the Simulated Educational Computer (SEDCOM), a hypothetical 377 (octal) word computer. SEDCOM is a small but powerful enough to provide an excellent introduction to machine language programing. The system, which is truly generative in that it creates and solves problems tailored to individual student needs, consists of four major components: a problem generator, a control unit, concept routines, and a SEDCOM simulator. Operationally, it proceeds by first acquiring student records and determining the amount of instruction needed. It then generates a sample programing problem and helps the student design a solution by dealing with sub-tasks. Concept routine guide the student through each sub-task and feedback continuously monitors progress. (PB)

The University of Connecticut
SCHOOL OF ENGINEERING
Storrs, Connecticut 06268

Computer Science Program

in the

Department of Electrical Engineering

A GENERATIVE CAI MONITOR FOR TEACHING MACHINE-LANGUAGE PROGRAMMING

by

Sumner E. Blount

This research has been supported by the Connecticut Research Commission Grant RSA 71-7 and the U. S. Office of Education Grant #OEG C-72-0895. The author also received support for this graduate education through a National Defense Education Act Fellowship.

May 1972

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. INTRODUCTION

This paper describes a computer-assisted instructional (CAI) system designed to teach and give practice in basic machine-language computer programming. The system has been implemented on an IBM 360 using the Conversational Programming System (CPS). Numerous 2741 remote terminals are available to the student so that access to the system can be achieved quite easily. For the sake of clarity, the system will hereafter be referred to as MALT - a MAchine Language Teacher.

The machine-language which is taught is that of SEDCOM, a Simulated EDucational COMputer as described in Booth [1]. SEDCOM is a hypothetical 377 (octal) word computer with single-addressing and one accumulator. It is quite similar in design to the Digital Equipment Corporation PDP-8 computer. All instructions in SEDCOM are identical to their counterparts in the PDP-8 instruction set with the exception of the I/O commands.

SEDCOM was used in this research because it provides the student with an excellent introduction to machine-language programming. Many computer science courses assume a knowledge of programming on a small machine and SEDCOM provides just such a background. It is also powerful enough to be used for non-trivial projects, yet small enough in memory size to be applicable to a CAI tutorial environment.

Review of CAI Research

Most efforts in CAI during the past few years have been devoted to creating systems which are more general in their applications and

more powerful in their problem-solving abilities. This trend has quite naturally led to the development of generative CAI systems. Such systems have many important differences from their predecessor, frame-oriented CAI. Frame-oriented CAI was inherently inefficient because all questions and possible student responses had to be specifically determined and programmed by the course author. As the student progressed through a series of questions, the system would follow a tutorial path which was pre-set by the author. Each response would cause a program branch and another question would be asked. There was very little ability to vary the material presented according to the needs of the student. There was also much inflexibility because additions or revisions in the system would often require massive amounts of re-programming. Frame-oriented systems basically allowed the computer to function only as a glorified filing cabinet in that it presented material to the student in a pre-determined, non-adaptive way.

Generative CAI systems, on the other hand, have allowed the material presentation to be more nearly geared to the student's abilities. Each user receives a variety of questions which closely reflects his current achievement level. As his performance improves, the system exercises less control and allows the student to perform more as an autonomous unit. Siklóssy [2] provides an excellent discussion of the major differences between generative and frame-oriented CAI.

Generative systems also allow a much greater flexibility because questions and problems are determined by the system instead of the course author. Modifications to the system become much less of an ordeal than for the frame-oriented system. The range of possible sample problems given to the student can be expanded quite easily by

merely adjusting certain parameters in the generation heuristics. However, changes in the general type of problems generated are very difficult for either type of system. Each system is usually designed as a vehicle for teaching a specific subject or type of problem, so changes characteristically involve major system reorganization.

Koffman [3] describes the implementation of a generative CAI system designed to teach an entire course in introductory computer science. Each student receives instruction in a series of basic concepts such as binary arithmetic, design of sequential circuits, minimization of digital networks, or the like. The system is constantly analyzing the student's performance to help select a suitable sequence of concepts for him. It can also function solely as a problem solver for use by graduate or advanced under-graduate students.

Research in Teaching Programming

Recent research in teaching computer programming has focused only on higher-level languages [4,5]. This is because these languages provide an excellent introduction to the basic concepts of computer programming. They possess certain characteristics, however, which limit their effectiveness in a CAI environment. Because of their complexity, it is impossible for the instructional system to determine the relative correctness of a student program. The program may be actually executed or simulated but if the results are incorrect, the cause and location of the program error cannot be established. The system may be able to provide much instruction in basic programming concepts such as the use of variables, loops, subroutines, etc., but its effectiveness ceases when the student finishes designing his program. Since the

system has not monitored the development of that program in any way, it is powerless to make judgements as to the program's effectiveness and location of any errors which might exist. The undesirable net result is that the student is expected to judge the merits of his own program.

Although this simulation procedure might be partially acceptable for higher-level languages, it is unsuitable for machine-language instruction. This is because programs written in machine-language are more susceptible to minute errors in logic than are other types of programs. Extreme care must be taken because, for example, a successful program execution does not by any means guarantee that a second trial will produce identical results. The initial state of the machine may have a profound effect on the final outcome whereas this is not usually the case in dealing with user-oriented languages. Consequently, it is better to monitor and verify the student's solution program as it is being designed.

Machine-language was chosen for this research also because it provides a more structured environment. A particular algorithm may be implemented many different ways in higher-level languages while in machine-language, fewer such possible solutions exist. This fact allows the system to monitor a user program and isolate errors as they occur.

Feurzeig and Wexelblat [4], have developed a system called SIMON as a vehicle for teaching simple programming for use in the fields of mathematics, physics and engineering. After being supplied with a sample problem, the student attempts to design a solution program. SIMON then tests this program against a "true" program

supplied by the author. If the results differ, the system is unable to help correct the error aside from determining if any program variables are inappropriate to the computation. Although this may in some cases be the cause of the error, it is likely that more program analysis is necessary to correct the situation. Also, each problem used by SIMON is specifically written and the solution programmed by the course author. The system can neither generate nor solve its own problems.

Fenichel and Weizenbaum, [5], have also done extensive research into computerized programming instruction. Their system, called TEACH, is quite complex and is capable of teaching an entire course in programming concepts. The course presently consists of eighteen chapters, each one introducing the student to new programming principles. Although the system appears to be quite effective and successful, Fenichel freely points out that it attempts only syntactic program corrections and leaves the semantic analysis and correction of logical errors to the student.

Norton and Slimick [6] describe a system which is most similar in design to MALT. The two primary differences are that their system uses a simple assembly language instead of machine-language and it makes no attempt at analyzing student program errors. Their complete system is composed of three major components: a "driver" which serves as a student-curriculum interface, a simulated machine very similar to SEDCOM, and an interpretive assembler. As the student types in each assembly language statement it is coded in absolute form and checked for syntactic correctness. Upon completion, the program is "run" and the system determines the results which should have been

obtained. The student is responsible for determining not only the existence but also the location of any program errors.

The systems of both Norton and Fenichel have several similar properties. Both provide the student with extensive instruction in many different areas of programming. Though Norton focuses heavily on a specific language, each system primarily strives to teach programming concepts of a general nature. This is very difficult in the MALT system due mostly to the simplicity of the SEDCOM language. It was decided that the most effective means of teaching the language was to actually guide the student through the use of it.

Program Generation and Verification

Considerable effort has also been devoted in recent years to the dual problems of automatic program generation and proof of program correctness. Manna and Waldinger [7] have developed a successful algorithm for program generation. First, various program parameters are established such as an input vector $\bar{x}$, input predicate $\phi(\bar{x})$, partial function $z = f(\bar{x})$, and output predicate $\psi(\bar{x},z)$. $\phi(\bar{x})$ and $\psi(\bar{x},z)$ are conditions which must be met if the input and output vectors are to be considered appropriate to the process. The function $z = f(\bar{x})$ is precisely the operation which the program is intended to perform.

By deriving and proving theorems based on these quantities and their relationship to the output predicate, a pr gram can be constructed which implements the function $f(\bar{x})$. These results are quite interesting because they represent an amalgamation of much work done in related areas such as mathematical analysis, theorem proving and set theory applied to the relatively new field of program generation. The concepts represented by Manna's work were found to be valid and useful in the

implementation of the MALT system. This will be explored more fully in Chapter 6.

Overview of MALT System

The purpose of MALT is to teach the various techniques which are used in programming. Inherent in this task, however, is the ability of the system to determine when an error has been made in the student's program. MALT attempts, through constant monitoring of the student's program, to determine not only the existence of logical errors, but also their location in the program. This ability enables the system to be much like the human teacher; that is, it can note and correct logical errors before they develop into undesirable programming habits.

MALT is a generative CAI system in two important senses. First, it creates its own sample programming problems using a variety of heuristic techniques which will be discussed later. It is not dependent upon the course author for a complete supply of ready-made problems. Instead, by beginning with only a series of basic problem elements or sentences, it generates a problem which is consistent with the user's present ability. Also, each problem contains several variable program parameters which are generated randomly by the system. The result is that the number of possible different problems a student may receive is virtually limitless.

Another important way in which the system is truly generative is its ability to design a solution program for the problem which it has generated. By using basic algorithms supplied by the course instructor, the system can produce the actual machine code of a solution program. This implies that the system is quite flexible since later alterations and extensions involve only the addition of new programming algorithms,

not massive system reorganization.

The system attempts to tailor its presentation to fit the abilities of the student. Any problem which is generated is designed to provide the student with a challenge while not being beyond his capabilities. The dialogue initiated by the system will also be governed by the user's performance. A beginning student will receive a wide variety of hints and suggestions for the design of his program. Also, his errors will result in quite explicit and complete remedial messages. As the student progresses through the material he will receive less system information and be given more freedom in his programming actions. When the student achieves high proficiency, the system can function purely as a problem-solver in that all programs are generated by it. This facility is useful if a student desires to study examples of advanced problems and their corresponding solution programs.

As the system questions the student, it is constantly developing its own solution program for comparison with the student's program. It is also continually updating its knowledge of the status of the user's program. In this way a given programming concept is rarely presented the same way more than once to a particular student. The student's enjoyment of the system is thereby greatly enhanced because he receives new dialogue with every problem.

Summary of System Operation

The actual operation of MALT is straightforward. After the student identifies himself to the system his records are obtained and evaluated. These records determine the amount of instructional guidance which the student will receive. Next, a sample programming problem suited to his abilities is generated. To help him design his program,

the system will then develop a logic chart, or list of "sub-tasks". These sub-tasks break the problem into a series of smaller, more manageable steps and are of great help to the novice programmer.

As each sub-task is reached in the programming process, a corresponding Concept Routine is entered which will guide the student through the construction of that part of his program. During this phase the student is constantly being given feedback as to the correctness of his program. If his program introduces logical errors, the system will point these out and offer helpful suggestions for their correction. If the system feels that the student might benefit from observing his program in operation, it also has the capability to simulate program execution.

The system is constantly evaluating the student's performance and updating his permanent file. This is necessary because his achievement determines not only the difficulty of the problems given him, but also the amount of interaction which he receives during the design of his program.

This research effort has been intended primarily to overcome the weaknesses in previous systems as described above. However, it is hoped that the MALT system will be complimentary, not contradictory, to earlier efforts. In some respects, the structure of the present system is much more restricted than others. However, the techniques and heuristic methods used in the system have wide applicability and represent a significant basis for possible further research.

Although the entire MALT system is quite large and complex, it can be considered as a series of interconnected modules. Each such module will be discussed in the following chapters. An overall view

of the system organization is given in Chapter 2. Chapters 3 through 5 examine each major component of the system in detail. These include the Problem and Logic Generator, Control Unit and the array of Concept Routines. Chapter 6 discusses the program verification and simulation abilities of the system. The student records and their evaluation are considered in Chapter 7. The final chapter considers the significance and possible expansion of the MALT system.

II. SYSTEM OPERATION

The operation of the MALT system has been designed to be as flexible as possible. Although certain obvious restrictions must always be placed upon the user when teaching programming in a tutorial environment, the student nonetheless has been afforded maximum freedom in the operation of the system. This enables the system to satisfy the requirements of any user regardless of whether he is a beginner desiring in-depth instruction or an experienced programmer wishing only a moderate review.

Level of Competence

The principal parameter which determines the amount of system interaction which the student receives is his LEVEL. This is a basic system variable and is the only parameter which is utilized by every system component. It determines to a large extent not only how much instructional dialogue is initiated by the system, but also how much freedom the user is allowed in the designing of his solution program. It also determines the difficulty of the problem which a student receives.

The value of LEVEL is always in a range from 0 to 2.5. Upon a student's first use of the system, his LEVEL is set to a value of .3. This allows the student a certain margin of freedom because it is very possible that he might make several mistakes as be begins to acquire facility with the system.

As the student increases his use of the system, his performance

LEVEL will certainly tend to rise. This is practically assured because the problems which are given to a beginning student are quite low in difficulty. If a student's LEVEL is consistantly in the 0-.5 range for a long period of time, then either he is completely unsuited for even beginning programming instruction or he is deliberately exhibiting sub-standard performance. In either case his use of the system will give him negligible benefical results.

During system operation the student will gain experience in many different types of machine-language programs. As his programming ability increases, so his LEVEL will also rise. This gradual increase in the LEVEL is determined by his answers to system-generated questions and by how well his solution program performs the functions assigned to it. The system will tend to initiate more dialogue with a beginning student than with one who has gained some facility in programming.

The system will also place a few more restrictions on the beginner in terms of the amount of freedom he is allowed in writing his program. The philosophical basis for this design is that although programming is definitely a "learn by doing" process, there are certain concepts and techniques in basic programming which should be taught. Once the student has been instructed in techniques which are generally considered to be "good" programming habits, it is hoped that he will continue to use them in later programming efforts. However, as he progresses through the instructional sequence he becomes, in general, free to use his own techniques.

The rate of progression through the programming material is not fixed. It is obvious that while two students may be subjected to equal numbers of system questions, the difficulty of these questions

may vary drastically. This fact implies that all questions should not be weighted equally by the system in the calculation of the student's current LEVEL. Therefore, less difficult questions are afforded considerably smaller importance than ones requiring more programming ability or knowledge.

This variation of question weights is carried still further. Even though a certain question may not affect a student's LEVEL to a large degree, the magnitude of the LEVEL change depends on the correctness of the response. As an illustration, a student who responds correctly to a question should have his LEVEL increased a relatively small amount. However, an incorrect response to the same question tends to indicate that the system is operating above the student's programming ability. In this case, his LEVEL should be decreased a significant amount to place him more in line with his ability.

When the student reaches a LEVEL of 2.5, it is assumed that he has gained maximum facility in machine-language programming. Therefore, the system will henceforth function only as a problem-solver in that it will completely generate all programs for the user. This facility is very useful to students who would like to observe examples of more difficult problems and the structure of their solution programs. Students at this LEVEL should be able to derive a great deal from this type of self-study. If the student still wishes to be quizzed on his solution, he may press the console ATTENTION (ATTN) button and request that his LEVEL be lowered as will be explained later.

All the techniques described are designed to progress the student through the material at a maximum stable rate which is consistent with his abilities. Since erratic fluctuations in the LEVEL

are virtually impossible, it appears that this goal has been achieved.

Another method used by the system to tailor its presentation to the ability of the user is to selectively alter the value of a student's performance indicator, (PERF), depending on the difficulty of the problem. This value represents the student's performance on each of the three segments of each sample problem. It is added to his current LEVEL only after the programming of each such primitive. As the student programs each segment of his problem, his performance is evaluated. This PERF value is then divided by a stability factor to ensure that erratic changes in a student's LEVEL do not occur. The stability factor is considerably smaller if the student's LEVEL is less than 1.5 then if it is greater than this value. The end result of this process is that a typical student will progress quickly through the simpler problems and tend to move more slowly through the more difficult ones. This feature also insures that the student will progress through the problems in an orderly fashion. The material is adjusted according to the student's performance but there is no erratic change in presentation because a given question might or might not have been correctly answered.

The technique of gradual advancement also lends itself nicely to system alterations. If class results indicate that progress through the material is not at the proper pace, a change of only one statement in the Control Unit can remedy the situation. By selectively altering the value of the stability factor which is divided into PERF, the proper pace can be achieved without having to change all Concept Routines. This facility has been used quite frequently and it is felt that a reasonable presentation speed has now been achieved.

During the programming of any problem it is inevitable that certain tasks are required of the student which seem quite easy to him. If the student is required to perform these tasks completely with full system interaction, considerable boredom and restlessness will result. Therefore, if the system feels that a particular task, or concept, is too easy for the student, it will write the program segment which performs the given task. In this way, valuable time is not wasted on trivial or meaningless exercises.

System Components

There are four major components in the MALT system, each of which will be discussed fully in later chapters. Figure 1 provides a block diagram of the general system organization.

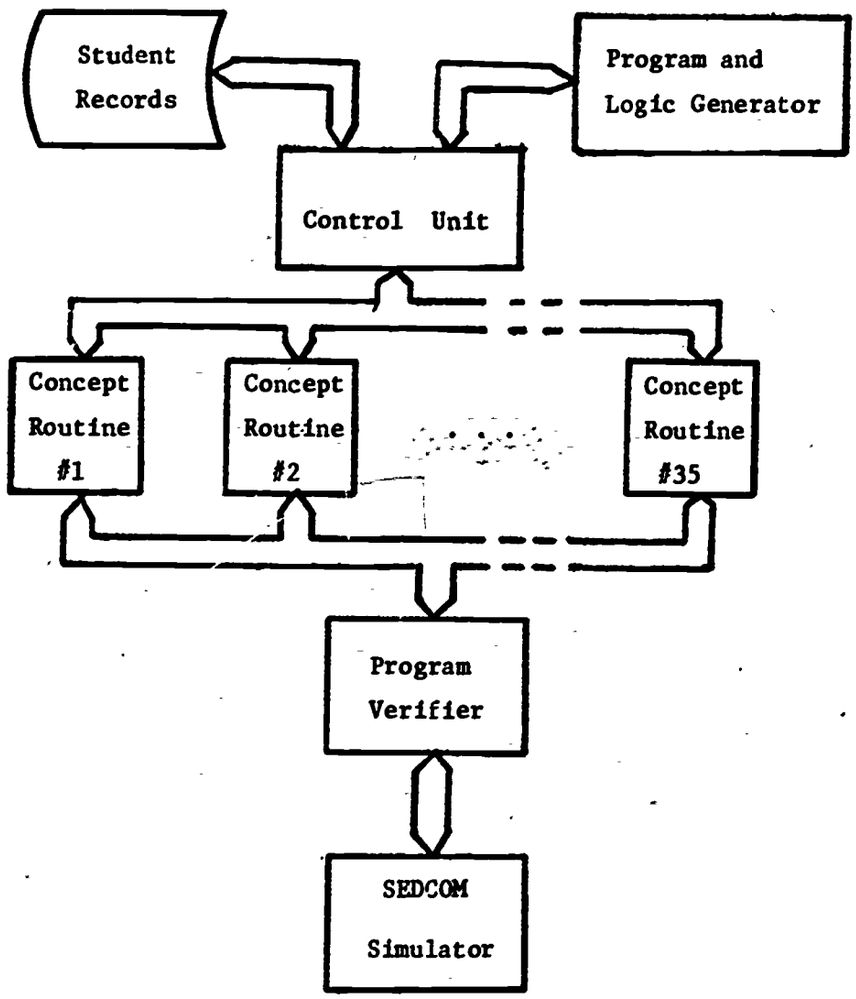
The first component is called The Problem Generator and is responsible for constructing a problem and a list of "sub-tasks" representing the logic necessary to solve the problem. The Problem Generator consists of five system programs: one to generate the problem and its parameters (gener), one to store the text for the printing of the problem (prob), and three to generate the program logic (chart 1, chart 2, chart 3).

The second component to be considered is the Control Unit. It is responsible for the orderly coordination of all system activities. It determines and maintains records of all parameters which will be needed in the user's program. From the Control Unit, each Concept Routine is called to help the student design that segment of his program.

Every fundamental concept or sub-task of machine-language

Figure 1

Block Diagram of System Organization



programming is represented by one of the Concept Routines. Each routine is composed of three phases: a generation, a teaching, and a remedial phase. The generation phase has the ability to actually write a program to solve the appropriate sub-task. The teaching phase is concerned only with helping the user construct his own program. The final phase is a collection of remedial statements and information to help correct user errors.

The last major system component is the SEDCOM simulator. This package is used whenever partial execution of the user's program is desirable. Associated with the simulator is a Program Verifier which enables the system to make detailed analyses as to the correctness of the program and nature of any existing errors. The simulator is used only in isolated instances but nevertheless provides the user with substantial insight into the programming process.

Modes of Operation

To enable MALT to satisfy the diverse desires of many users, several modes of operation have been developed. There are two modes reserved for student use and one mode for use by the instructor. The primary mode is called the Student Mode and is used whenever the student is enrolled in the appropriate computer science course. In this mode, the student receives all facilities of the system and his performance is monitored and stored in his permanent record. He is afforded no choice concerning which system functions he may receive. He is provided with a sample problem, a logic diagram consisting of a sequential listing of the problem sub-tasks, and tutorial aid in the designing of a solution program. If his performance has been exceptional, however, the system will function as a problem-solver as described above.

If the student is not currently enrolled in the course, the system operates in the Non-Student Mode. This mode differs primarily in that the student is not required to follow the usual instructional sequence of the system. For instance, he may receive a problem of any difficulty level, an appropriate logic chart, a solution program, or any combination thereof. He may also receive tutorial instruction exactly as if he were in the Student-Mode.

A second difference in the two student modes is that the student's performance is not permanently stored if the system is in the Non-Student Mode. Although his performance is monitored throughout, it is not saved for later reference because a permanent file has not been established for him. However, his performance on general programming concepts is included with that of the regular students. In this manner, the overall record of student achievement in machine-language programming is made more statistically representative.

If at any time during system operation a user wishes drastic changes in, or termination of, the presentation of the material he may press the Attention (ATTN) button on the 2741 keyboard. This action causes a routine to be entered which determines the user's wishes. If he indicates that he wants to terminate system operation completely, a flag is set and control is returned to the Control Unit. His records are then updated and operation halts. If however, he indicates that he would like to progress either at a faster or slower rate, his LEVEL is adjusted accordingly. The change remains permanent regardless of his performance at the new level. This enables each student to progress at a rate which he feels is appropriate, not one dictated by the instructor or even the MALT system itself. Also,

his performance is readily available to the instructor in the form of a percentage score, so he cannot influence his grade for the course by selectively altering his own LEVEL.

The final mode of operation is the Instructor Mode. It is entered by supplying a keyword to the system which can be changed periodically by the instructor. In this mode, the instructor has several options available. He may receive a class list, a record of the cumulative achievement of each student, a complete history of every student's performance, or an evaluation of class achievement on each of several broad programming concepts. This flexibility enables the instructor to observe not only general areas of class weakness but also specific difficulties of any individual.

Finally, the instructor may reset any or all student files if they begin to fill up. The system requires a special password for this operation so that unauthorized access to student records is virtually impossible. Figure 2 illustrates the use of the Instructor Mode.

The system tailors its operation to the needs of the user through the use of a numerical array called FLAGS. These values are set to indicate many various conditions, primarily reflecting the current mode of system operation. The array greatly facilitates the task of system alteration because such changes involve merely adjusting the initial value of a particular array element. Also, the number of possible conditions which can be sensed by the system greatly enhances its overall flexibility. Table 1 indicates the current values and significance of each element of the FLAGS array.

Constraints of CPS

There are certain constraints placed upon the operation of MALT

Figure 2.

PRINT-OUT OF STUDENT RECORDS

IN THE INSTRUCTOR MODE

YOU ARE NOW IN THE INSTRUCTOR MODE.
DO YOU WANT A CLASS LIST?
_no

WOULD YOU LIKE TO SEE THE CLASS PERFORMANCE RECORDS?
_yes

NAME	LEVEL	# OF USES	TIME	RESPONSES	ERRORS	SCORE
SBELL	1.7	07	087	0075	008	89
JTOKARSK	2.5	07	122	0090	018	80
HSPRECHM	2.3	09	155	0180	014	92
EBIASE	1.2	03	037	0030	003	90
BSCHROTE	1.2	05	134	0107	026	75
JKWOLEK	2.4	07	103	0080	010	87
MSEGER	1.7	08	155	0135	014	89
BCOTTLER	2.1	07	156	0102	025	75
AWESBECH	1.9	04	201	0078	013	83
PKIANG	1.1	02	034	0028	001	96
BSHELDON	2.2	07	126	0091	017	81
JKALINOW	1.1	04	094	0031	007	77

CLASS AVE= 84.5 STAN. DEV.= 6.68954

Table 1.

VALUE OF FLAGS ARRAY

SUBSCRIPT	VALUE	MEANING
1	1	STUDENT WANTS ONLY PROBLEM
	2	STUDENT WANTS PROBLEM, SUB-TASKS
	3	STUDENT WANTS ALL FACILITIES, NSM
	4	STUDENT WANTS ALL FACILITIES, RSM
2	1	INSTRUCTOR WANTS CLASS LIST
	2	INSTRUCTOR WANTS CONCEPT RATIOS
	3	INSTRUCTOR WANTS STUDENT FILES
	4	RESET ALL STUDENT FILES
	5	RESET ONLY "PROBS" FILE
3	0	REGULAR STUDENT MODE
	1	NON-STUDENT MODE
	2	INSTRUCTOR MODE
4	-	BLANK FOR FUTURE EXPANSION
5	-	BLANK FOR FUTURE EXPANSION

by the very nature of the CPS system which deserve explanation. The CPS system operates in an interpretive mode and is executed in low-speed core on the IBM 360/65. This results in considerable delays in program execution speed and system response time. As the number of people using CPS at any given time increases, so also do the delays in operation increase. At certain times these delays become virtually intolerable especially for someone who is not familiar with CPS operation. However, normally response times are about 3 seconds which appears to be quite acceptable.

Another restriction placed upon the system by CPS is that of storage space. The maximum size for any given CPS program is four pages, where each page represents four-thousand bytes of core storage. Also, each user is allowed a maximum of twelve active pages in CPS core at any given time. Since the control unit occupies four pages, there can never be more than eight additional pages active. This limitation has been reached many times and corrective measures must be taken whenever it occurs. It appears that system implementation could have been considerably simplified if this restriction did not exist.

A final problem inherent in the structure of CPS is the possibility of computer system failure. A partial solution is the fact that the system updates most of a student's records after each problem solution. Nevertheless, much time and effort are usually wasted in the event of a system failure.

Despite these limitations the CPS system performs well in an interactive CAI environment. The process of interactive dialogue with a terminal usually proves quite fascinating and enjoyable to the beginning student. It is hoped that possible future system improvements

in CPS may remove some of the above restrictions. Such improvements might include the use of high-speed instead of low-speed core storage, or the allocation of more CPU time to the CPS system.

In all later discussions of the structure of SEDCOM, it should be emphasized that while SEDCOM itself is obviously a binary (or octal) machine, the CPS system operates only in the decimal mode. This fact necessitates that all SEDCOM variables be stored internally as character strings rather than numerical quantities. Although this situation poses no great programming problems, it has a profound effect on the length and operating time of any system program. For example, the trivial act of incrementing a four-digit octal number requires considerable time and character string manipulation to accomplish. If this important limitation did not exist, a greater amount of time could have been spent on expanding the overall scope of the MALT system.

Now that the basic operation of the system is apparent, it is necessary to discuss each of the system components in turn. The first to be considered is the Problem and Logic Generation.

III. PROBLEM AND LOGIC GENERATION

The primary area in which the generative capabilities of MALT are most evident is in the construction of sample problems. As explained in the introduction, most previous systems resort to storing entire problems which have been supplied by the course instructor. This technique is wasteful of disk storage and contradictory to the principles of generative CAI.

MALT employs a much more sophisticated as well as interesting heuristic for generating its problems. Instead of storing a problem in its entirety, only parts of it are kept. These segments are called "problem primitives" because they are like basic building blocks which the system uses to design its problems. They can be thought of as representing either a first, second, or third sentence of a typical machine-language programming problem.

Each group of problem primitives is designed around a different programming concept. The first such group, herein called "first problem primitives", typically is concerned with methods of data input into a user program. This involves only the use of SEDCOM's reader but the type of input data may vary greatly. A student may be required to construct a program to do such tasks as reading in a series of characters and storing them in memory, or reading a series of multi-digit numbers and forming their numerical value.

The next group, second problem primitives, deals exclusively with the processing of data which is resident in the computer memory. This group obviously encompasses a vast number of possible problems and

is therefore the largest group of primitives. The beginning student will be required to perform trivial processing such as adding or subtracting the contents of two memory registers. More difficult processing includes program decisions based upon the contents of the accumulator or link, operations involving large arrays of memory registers, and the like. Finally, the advanced students will be required to perform quite complex algorithms such as extracting a particular symbol or numerical quantity from a table which has been stored in memory.

The third group of problem primitives deal exclusively with various methods of computer data output. Because of the simplified structure of SEDCOM, this may involve only the operation of a teletype printer. However, like data input, it may take any one of many possible forms. The student may be asked to print out such things as his own name, a particular message, a register address, or the contents of a register. These operations normally involve quite advanced techniques and provide the student with excellent practice in many areas of machine-language programming.

There are presently eight elements of the first problem primitives, fourteen members of the second group, and ten elements of the third. Tables 2 through 4 provide a list of current problem primitives. Each underlined quantity represents a parameter (to be discussed below), which is generated randomly for each problem. To construct a typical problem for the student, the system selects an element from each group according to certain constraints and combines them to form a unified, meaningful problem.

To allow for less difficult problems, each group has one element called a null problem primitive. These can be generated whenever a

Table 2.

FIRST PROBLEM PRIMITIVES

1) Read in a series of ASCII characters ending in a "*" and store them starting in location 100.

2) Read in a series of ASCII characters which end with a "X" and keep a counter of the number of "S"s which occur.

3) Assume a table has been set up starting at location 100 consisting of a 3-character symbol followed by a number; So each entry takes up 4 registers and there will be 07 of these entries.

4) Read in a series of 4-digit numbers and store their value starting at location 250. The input will end when the first character of a number is a "X".

5) Read in 24 (octal), 2-digit numbers and store their value starting at location 311.

6) Read in 31 ASCII characters and store them in locations 220 thru 251.

7) Null primitive

8) Null primitive

Table 3.

SECOND PROBLEM PRIMITIVES

1) Search registers 220 thru 250 for the 1st instruction which begins with the octal digits "70". (EXAMPLE: 70XX)

2) Search registers 125 thru 160 for the largest number.

3) Search registers 200 thru 230 and stop at the 1st register which contains a positive number.

4) Search registers 160 thru 220 for the octal number "7402".

5) Form the absolute value of register 140 in the Accumulator.

6) Add the contents of register 150 to the contents of register 170.

7) Subtract the contents of register 160 from the contents of register 210.

8) Multiply the contents of register 211 by the contents of register 310.

9) Form the 2's complement of the contents of register 304 in the Accumulator.

10) Form the sum of registers 170 thru 220 in the Accumulator.

11) Compare the contents of registers 120 and 210. Place the smallest in location 255 and the largest in location 260.

Table 3. (Cont.)

12) Null primitive

13) Search the table for the symbol "JMP" and retrieve the corresponding number. If it is not in the table, then halt the program.

14) Null primitive

Table 4.

THIRD PROBLEM PRIMITIVES

1) Print out the register number, 3 spaces, and the message, "POSITIVE".

2) Next, deposit this number in location 305.

3) If this results in a non-zero LINK, stop with the (ACC)=7777, otherwise stop with the (ACC)=0000.

4) Finally, print out the 4-digit contents of the Accumulator.

5) Lastly, print out the register where it was found, 2 spaces, and the contents of that register.

6) Print out the symbol, 5 spaces, and the register number where it was found.

7) For registers 220 thru 250, print out the register number, 4 spaces, and the octal contents of that register.

8) Null primitive

9) Print out the message, "HELLO".

10) Print out your own name.

simpler problem dealing with only one concept is desired. A beginner can then receive an easier problem consisting of, for example, only a processing primitive. Null primitives also effectively increase the number of possible problems which can be generated because each problem does not have to consist of three complete primitives. By the selective inclusion of null primitives, the student feels that he is receiving a much wider variety of problems although this is not strictly the case.

Problem Generation Techniques

There are three primary ways in which the system adapts its problem generation techniques to the user. Although these generation constraints serve primarily to limit the number of possible problems for a given student, the end result is a meaningful problem which is consistent with the user's ability.

The most important constraint heuristic is that of problem plausibility. Each problem must be reasonable in that it requires the student to perform relevant tasks. To generate a problem which involved contradictory or totally unrelated operations would clearly be undesirable. Therefore, all problems are generated so that each of the constituent primitives are logically related and the result is a plausible, meaningful problem.

The implementation of this heuristic implies that not all combinations of primitives are possible. Each primitive can be combined only with a subset of the following primitive group. Such a subset typically includes only those primitives which would form a valid problem when combined with the primitive previously selected. For some primitives

this subset is quite small while for others, notably the null primitives, it is as large as the entire next group. Within these plausibility bounds, however, primitive generation is totally random.

A second constraint on generation is the obvious factor of a student's past performance. It is important that the user receive a problem which is maximally consistent with his current programming ability. For this reason, each problem primitive is assigned a pair of values, (α,β), called a threshold pair. The user's current LEVEL must fall between these values if this primitive is to be used. If his LEVEL is outside this interval the primitive would be either trivially simple or hopelessly difficult and therefore counter-productive to his learning process.

These threshold pairs are set by the instructor and reflect his judgement as to the difficulty of the particular problem. They are easily changed, however, to adapt to differing class abilities. Most of the simpler problems have a threshold pair of approximately (0,1) while the more difficult ones have a pair of $(1.5, \infty)$. This is only an estimate though, because very few primitives have equal threshold pairs (see Table 5). As the student progresses through the material, the advanced primitives become more likely to be generated and the easier problems are excluded from possible generation. Each problem is therefore appropriate to the student's current LEVEL yet varied enough in difficulty to provide interesting combinations.

A final restriction placed upon generation is intended to avoid repetitious problems. After an entire problem is constructed, the students's past history is consulted to determine if this problem is the same as one previously solved. If so, the problem is rejected and

Table 5.

PROBLEM PRIMITIVES AND THRESHOLD PAIRS

FIRST PROBLEM PRIMITIVES	THRESHOLD PAIRS
1	(1.0, ∞)
2	(1.5, ∞)
3	(2.0, ∞)
4	(1.6, ∞)
5	(1.2, ∞)
6	(0.5, ∞)
7	(0.0, ∞)
8	(0.0,2.0)

SECOND PROBLEM PRIMITIVES	
1	(1.0, ∞)
2	(1.2, ∞)
3	(1.1, ∞)
4	(0.5, ∞)
5	(0.0, ∞)
6	(0.0,1.0)
7	(0.0,1.5)
8	(0.0,1.5)
9	(0.0,1.3)
10	(0.5, ∞)
11	(1.0, ∞)
12	(0.0, ∞)
13	(2.0, ∞)

THIRD PROBLEM PRIMITIVES	
1	(1.1, ∞)
2	(0.0,2.2)
3	(0.0, ∞)
4	(1.0, ∞)
5	(1.5, ∞)
6	(2.0, ∞)
7	(2.0, ∞)
8	(0.0, ∞)
9	(0.0,1.7)
10	(0.0 1.7)

another is generated. Because of the large number of possible problems available, it is important that ones which are not only identical but even similar to previous ones be rejected. Therefore, if the second primitives and either the first or third ones are identical to the respective primitives of a previous problem, the problem is judged as being repetitious and discarded. The second primitives are regarded by the system as being most important due to their large number and the fact that they form the basis of most programming problems.

If, after several attempts, the system does not generate a new problem for the user, it accepts the current one and informs the user that it may be similar to a previous one. This saves possible wasted time in problem generation and eliminates the threat of infinite loops.

All of the preceeding techniques are designed to provide the student with a wide variety of meaningful problems. In the current system implementation, there are approximately eighty completely different problem formats which are possible. There is therefore small chance of complete problem duplication.

Once an appropriate problem has been constructed there are various parameters within it which must be included. These include such values as the address of a memory register, the address of a table, the amount of input data, a string of output text, etc. There are many of these parameters and their value will vary with each successive problem generation. The resulting situation is that although there are roughly eighty basic problem formats, the number of completely different problems which are possible approaches infinity. Clearly this illustrates the power and effectiveness of the problem generation heuristics.

Problem parameter generation is subject to only one minor restriction. The parameters are constrained to fall within certain reasonable intervals of core memory. This insures that parameters will not interfere with each other or the user's program sequence. If this restriction did not exist, user programs which were self-modifying might often result.

When problem and parameter generation is completed the problem is presented to the student. This often requires small amounts of sentence modification and generation in order to get a syntactically correct form. It usually involves only proper agreement between the sentence verb and modifying preposition so that forms such as: "Subtract x by y" do not result. These modifications are not extensive but they do illustrate the limits to which sentence generation can be taken. The next logical step in generative CAI research could be to construct problems not from sentence primitives as done herein, but from simple word primitives. The theoretical value of such an effort would be much greater but so also would be the difficulties involved. Table 6 provides several examples of typical problems which might be generated by MALT.

The various heuristics used by the system in problem and parameter generation are extremely important. They are flexible enough to provide a wide variety of sample problems and specific enough to tailor each problem to the present abilities of the student. The technique of building problems from basic primitive elements eases the programming task of the instructor and allows him to concentrate on other areas of system design. It also makes the overall system generative in the broadest sense of the word.

Table 6.

SAMPLE PROBLEMS

1) Read in a series of ASCII characters ending in a "X" and keep a counter of the number of "A"s which occur.

2) Form the sum of registers 242 thru 262 in the Accumulator. If this results in a non-zero LINK, stop with the (ACC)=7777, otherwise stop with the (ACC)=0000.

3) Read in 10(octal) ASCII characters and store them in registers 260 thru 270.
For registers 260 thru 270, print out the register number, 1 space, and the contents of that register.

4) Read in 24(octal), 3-digit numbers and store their value starting in register 300.
Search registers 300 thru 324 and stop at the 1st register with a zero number in it.
Print out the register number, 2 spaces, and the message "ZERO".

5) Form the absolute value of register 272 in the Accumulator.
Finally, print out the 4-digit contents of the Accumulator.

6) Print out the message "HI THERE".

7) Search registers 212 thru 230 for the largest number.
Next, deposit this number in register 310.

A valuable by-product of problem generation is that the solution to the programming problem is implicitly provided to the system. The system is informed of the sequence of problem primitives and parameters selected. This knowledge is used both to construct a logic diagram of sub-tasks and to monitor (and generate in part) the solution progra- This process is explained in the next section.

Concept Sequence

The final task of the Problem Generat on is to construct the Concept Sequence. This variable (called SEQ) is a character string consisting of a series of digits which represents the sequence in which the Concept Routinesmust be called in order to design a solution program. In general, every pair of digits represents the number of a different Concept Routine. Each Routine is assigned an aribtrary integer whose value falls in the range (1,35). A sequence of such numbers is assigned to each possible problem primitive by the course author which reflects the basic logical flow of an appropriate solution program. The Concept Sequence for the complete problem is then merely the concatenation of the sequence of each respective problem primitive.

To clarify this point, assume that a sample problem consists of primitiv s 3, 10, and 5. The Concept Sequences for these primitives are, respectively:

'2324 '

'23240503 '

and '323433 '

This implies that the solution program for the first primitive can be designed by successively calling concepts 23 and 24. The program for

the other primitives can be implemented in a similar manner. The Concept Sequence for the entire problem is therefore found by combining the sequence for each primitive yielding:

'2324...3433 '

The final element in each respective sequence is a null symbol which is used to indicate to the Control Unit that the end of a program segment has been reached. The end of the entire program is indicated by two adjacent null symbols in the sequence.

Almost all Concept Routines require only two digits in the Concept Sequence for their respective number assignment. However, it is sometimes necessary that additional parameters be passed in this string. As an example, the Concept Routine which teaches program branches must have available to it the destination of any such jump. Therefore, the two digits following its own code (which happens to be 26) represent the number of the Concept Routine to which the jump is to be directed. A SEQ value of '2603...' indicates that the Concept Routine should insert an instruction in the program to "JMP" to the beginning of concept number 3.

This technique of constructing a complete Concept Sequence from similar sequences for each primitive has proven to be quite effective. It is a highly compact method for representing the entire logical flow of a solution program. It also allows a wider variety of problems to be generated because each primitive is treated as a problem in itself. Since there is no logical overlap between such primitives, there is a larger number of combinations which can be made than if this method were not used.

Logic Generation

Another important generative facility in the system is its ability to extract the program logic from any given sample problem. This ability provides a structure within which student and system can operate effectively. It allows the student to visualize the basic flow of his program and eliminates much initial confusion and error. Also, the system requires that certain ground rules be established before a user designs his program. If no such restrictions were made, the system would, in effect, have to "understand" the user's program regardless of how he designed it. This is clearly an interesting but recursively unsolvable problem. For, if the system monitor could "understand", in some meaningful way, any program which was supplied to it, it could also interpret one which was a little more complex than itself.

This reasoning leads one to the obvious conclusion that since such a situation is unattainable, there must be some limits as to how the user may develop his program. It was decided that the best alternative was to show the student a flow chart of his problem and allow him to develop the program within the framework of this logic. Early results seem to indicate that students do not feel restricted by this method. In fact, they usually insist that it helps them greatly in learning machine-language programming because it aids them in structuring their program.

The Logic Generator is used after a problem has been presented and prior to the programming of any one of its primitives. It is also entered from the Control Unit after the programming of each

primitive to produce the logic for the following primitive. This allows each primitive to be totally independent from any other one and spares the student from long waits while large numbers of sub-tasks are printed out.

The function of the Logic Generator is fairly straightforward. It must scan the Concept Sequence and generate a list of logical sub-tasks for each concept in the string. The problem is more complex than this however, because many concepts require a variable number of sub-tasks depending on certain conditions. For instance, the concept which teaches the student how to input a series of numbers must be aware of whether there are a fixed or variable amount of these numbers. The list of sub-tasks generated for the latter case will be longer because various operations must be performed in the student's program to determine if the end of the input has been reached.

Even those sub-tasks which are similar for each problem will not be strictly identical. This situation implies that the Logic Generator is far more than a text buffer and printing routine. It must analyze both the parameters present and the Concept Sequence to determine the proper list of sub-tasks.

There is also a second complicating factor in this process. It is often necessary that a particular sub-task make reference to another one somewhere else in the list. For instance, the sentence "Jump down to sub-task 7" might occur as part of the logic for a particular primitive. The Logic Generator only knows, however, the relative point in the program to which this jump is directed. It does not know the exact number of the sub-task corresponding to that part of the program. It therefore must have a method of determining

the exact number of sub-tasks which are required for each element in the Concept Sequence. This will allow other parts of the program to be referenced by their appropriate sub-task number. As a result, each sub-task is made more specific and the entire list is much more coherent to the student.

This is accomplished in a manner analagous to a 2-pass assembler. The Concept Sequence is first scanned and a record is maintained of the beginning sub-task of every concept. A counter is incremented by the correct number of sub-tasks required for each concept. The result is stored in an array, Begin, as in the following example.

Let the Concept Sequence for a program segment be '08101107 '. The solution program can thus be written by calling concepts 8, 10, 11 and 7 in proper order. Now, let us assume that these concepts require 3, 1, 4, and 2 sub-tasks, respectively, to implement them. From this information we can determine the number of the particular sub-task which represents the start of any concept. In this case we have:

```
Begin (8) = 1
Begin (10) = 4
Begin (11) = 5
Begin (7) = 9
```

In subsequent logic generation, any sub-task may therefore refer to any other concept in the sequence merely by consulting the proper element in the Begin array.

The actual generation of the sub-tasks is performed next. The control program for the Logic Generator initializes a scanner to the beginning of the sequence string and sets a sub-task counter to one. Each concept number is then extracted and the proper list of sub-tasks

generated for it. Upon return to the control program, the sub-task counter and scanner have been updated to reflect this operation.

In the previous example, the sub-tasks for concept 8 would be generated first. The scanner would next be set at the second element of the sequence, "10", the sub-task counter would be incremented to 4, and processing would continue. The scanner is not advanced in the control program because each concept is not always represented by exactly two digits in the Concept Sequence.

When the first null symbol is reached in the string, the entire primitive has been reduced to a series of sub-tasks and control is returned to the system Control Unit. Figure 3 provides an example of a typical sub-task generation. Normally, the student would design his program for each primitive individually so he would not receive the sub-tasks for all primitives at the same time.

The techniques inherent in the system's problem and logic generation phases appear to be quite universal in application. They are, in general, not restricted to operation within the framework of the current system but instead are applicable to other areas of generative CAI research. The principle of constructing sample problems from smaller elements and adapting each problem to the ability of the student is basic to the tenets of generative CAI research.

Now that the techniques for problem generation are understood, it is important that the operation of the system Control Unit be considered. The next chapter provides a discussion of this relatively small but important component.

Figure 3.

EXAMPLE OF PROGRAM

SUB-TASK GENERATION

YOUR PROBLEM IS TO WRITE A PROGRAM WHICH WILL:

Read in 20 (octal) ASCII characters and store them in registers 240 thru 260.
Form the absolute value of the contents of register 240 in the Accumulator.
Finally, print out the 4-digit contents of the Accumulator.

Here are the sub-tasks for the 1st line
1) Initialize a ptr to register 240.
2) Initialize a ctr with the value of -20 (octal).
3) Read a character.
4) Store it away using the ptr.
5) Update the ptr.
6) Update the ctr and if it's not zero, jump back to start of loop.

Here are the sub-tasks for the 2nd line
1) Bring the number in register 240 to the Accumulator.
2) Check the sign of the ACC and if it's negative, then form it's 2's complement.

Here are the sub-tasks for the 3rd line
1) Store the contents of the Accumulator temporarily.
2) Set up a subroutine which will print a character.
3) Get the number to be printed and rotate it so the 1st (or leftmost) octal digit is on the right.
4) Mask out the left 9 bits, add 260, and call the print subroutine.
5) Do the same for the 2nd digit.
6) Do the same for the 3rd digit.
7) Get the number, mask it, add 260, and print it.

IV. CONTROL UNIT

The heart of the MALT system is a 4-page Control Unit stored under the name MAIN2. This component functions much the same as an operating system and is the only program which is always active in CPS core. It has access to all relevant information concerning the student and the current sample problem. It is designed to act as a system co-ordinator in that it sequences the student through the proper presentation of course material. A flow chart of the Control Unit is given in Figure 4.

The Control Unit has been designed to be as universally applicable as possible. That is, it is totally independent in the sense that it constantly monitors the status of the user's program regardless of the type or structure of that program. At any given time, the control unit is cognizant of all relevant user program parameters. In this respect, it functions not merely as a control program, but also as an "intelligent" programming monitor.

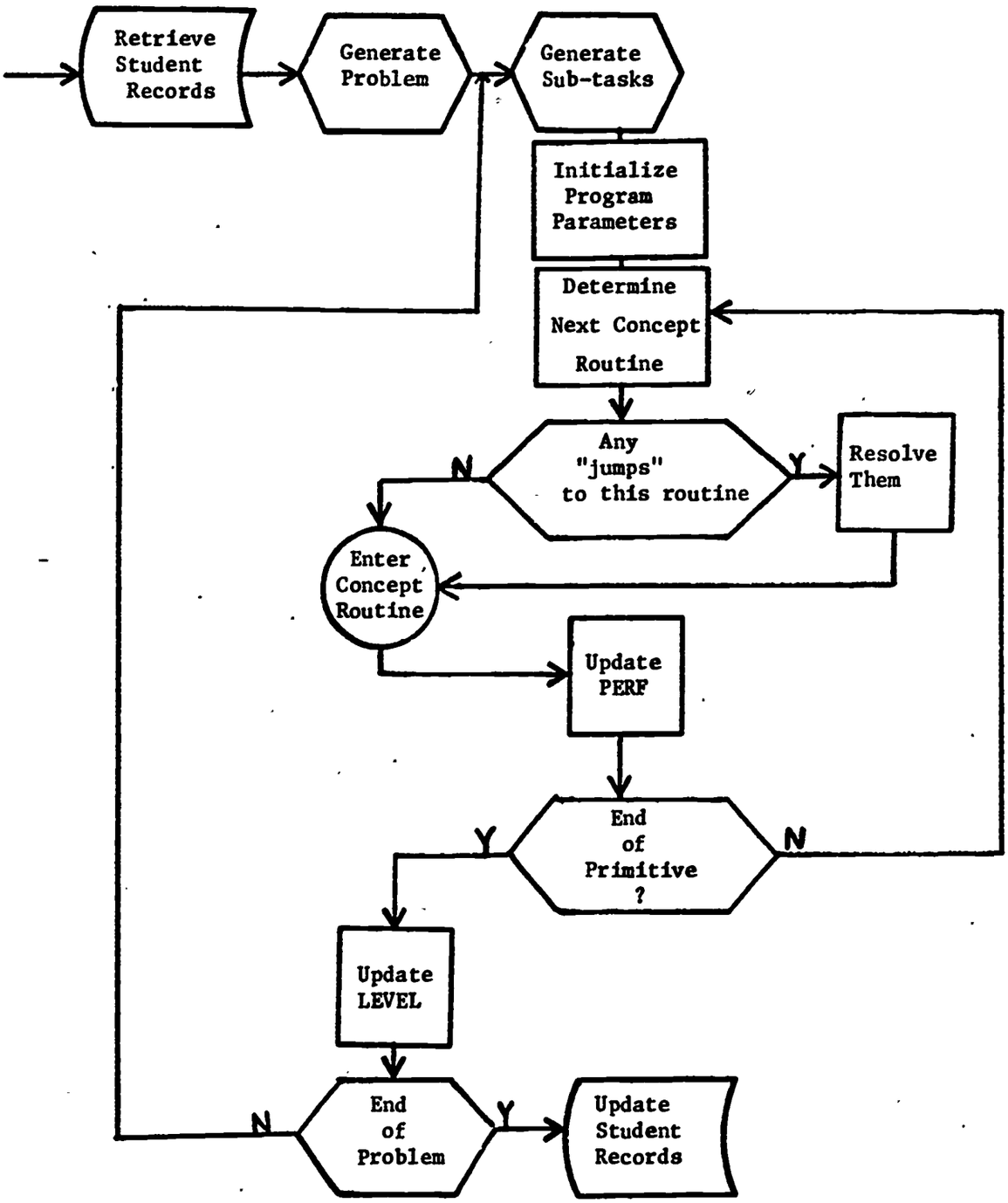
The most important aspect of the Control Unit is the way in which it records the various parameters of the user's program. There are two basic types of parameters: those which are necessary for all programs (universal parameters), and those whose existence depends only upon the current student program (program-dependent parameters).

Universal Program Parameters

After determining the user's name and securing his performance records, the Control Unit requests a sample problem from the problem

Figure 4

Flow Chart of Control Unit



generator. In doing so, it makes available all relevant parameters which might be useful to the generation algorithm. When control is returned, all parameters which the student will need in constructing his program are initialized. There are several of these which deserve special consideration.

The entire memory of SEDCOM is represented as a 377 element array, each element being a character string of length 4. This array is originally initialized to "0000" to insure that no previous programs cause errors in the current program. Since SEDCOM and CPS operate only in octal and decimal modes, respectively, all memory registers numbered with non-octal digits (such as registers 028 and 039) are ignored by the system.

As the student designs his program the location of the register which will hold the next sequential instruction is indicated by a variable called PC (Program Counter). For the sake of simplicity and to avoid possible overlaps with other storage locations, the program counter is initialized so that all user programs will begin in register "000". This variable is continually updated to insure proper instruction placement.

During the course of program development the student will usually require that numerical constants be stored in memory for later access by his program. These constants are placed by the system sequentially beginning in register 377 and extending downwards in memory as far as necessary. The current location of the end of this list is indicated by a variable called LIT (for LITeral pointer). As each new constant is needed, it is placed in memory at the end of the list and the LIT is decremented by one.

The technique of forcing placement of program constants at the top of memory was chosen for several reasons. The middle areas of memory (registers 150 through 350) are often reserved for lists and tables to be used by the student's program. To allow placement of the constants within the possible range of these lists would necessitate considerable alteration in memory allocation algorithms. Also, to allow the user to specify their location would introduce much needless dialogue. Finally, the arbitrary placement of constants in a program is basically irrelevant to the machine-language learning process.

Program-Dependent Parameters

The primary function performed by the Control Unit is to determine, at all times, the current status of the student's program. Since the system must be able to guide the student through his program development, it clearly must have a detailed record of every relevant program parameter. Some typical parameters which the system would need are the location of any pointers and counters, location and status of any program loops, the status of the accumulator, the location of any registers used for temporary storage, and the location of any unresolved forward jumps. This information does not directly affect the flow of the program but must nonetheless be available to the system to enable it to successfully monitor the student's program development.

A. Loop Pointers and Counters

The locations of pointers and counters are represented by the respective variables, PTR and CTR. These variables will exist whenever

there are loops within the student's program. In general, only one such loop will be active at any given time so that a series of nested loops rarely occurs.

These variables are placed along with numerical constants at the top of memory. Anytime a program necessitates the non-concurrent use of more than one loop, these same PRT and CTR registers are re-used as a method of instilling proper programming habits in the user.

The existence of a program loop is assumed by the system whenever a pointer or counter is initialized. The physical start of the loop is deemed to be the first memory register after this initialization process. By monitoring the beginning of a loop in this manner, the system can easily determine if the student correctly designs his end-of-loop decision sequence. The most common programming mistake of this kind occurs when the student attempts to jump back to the initialization sequence instead of the main body of the loop.

B. Temporary Storage Registers

During the construction of most programs, it is often necessary that some variables be given temporary storage in memory. The physical location of these parameters is usually invariant for the duration of the given program. However, their location is program-dependent so measures must be taken to record their location in any given program.

To implement this storage in MALT, an array of temporary locations (TEMLOC) was created. Each element of this array represents the storage location of a particular variable. This variable is stored temporarily, and

its location in memory is recorded in the TEMLOC array. Henceforth, all Concept Routines which require access to the variable need only refer to the proper array element. If any array element is empty, then the corresponding variable is either not relevant to the current program or has not been created as yet. Each concept routine can then revise its presentation to account for this situation. Table 7 is a list of all temporary parameter locations which are recorded by the system.

C. Accumulator Status

In machine-language programming, probably the most common source of error for the beginner involves the manipulation and current status of the accumulator. Since SEDCOM possesses neither a non-destructive deposit nor a destructive load instruction, the student and the system must constantly be aware of the present contents of the accumulator. For instance, if the accumulator is to be loaded with a given number, it first must be cleared of any previous contents. Likewise, a number which is to be used immediately and also saved for future use must be returned to the accumulator since the act of depositing it into memory destroys the accumulator's contents.

The contents of the Accumulator can be represented in two important ways. These two representations may often differ depending on the prior actions of the program. As an illustration consider the following program segment:

	CHARACTER SIGNIFYING END OF INPUT
3	NUMBER USED AS COMPARISON VALUE IN SEARCHING OPERATION
4	TEMPORARY STORAGE FOR ANY USER PROGRAM
5	LOCATION OF ANY SUBROUTINE TO PRINT A CHARACTER
6	CHARACTER IN INPUT STRING WHICH THE USER PROGRAM COUNTS
7	NUMBER WHICH IS TO BE OUTPUT ON THE PRINTER

At the highest interaction range the system performs less like a tutor and more like a general programming monitor. It does not demand that the student design his program in any particular manner besides adhering to the original sub-tasks. It asks very few questions and gives only a small number of hints and suggestions. The student is basically left on his own to see how well he can perform. He is informed if part of his program is in error but otherwise, the system does not require him to follow a particular program format. A sample list of concepts and interaction ranges is provided in the third column of Table 8.

A concrete example may help to illustrate these interaction ranges. The Concept Routine which teaches initialization of counters is divided into three such degrees of interaction. At the lowest range the student is told what the value should be which will be used to initialize the counter. A student in the second range will be told which values are possible but not which one he should use. At the highest plateau the student is provided with no suggestions whatsoever, (Fig. 6).

Location	Instruction	Comment
010	TAD 100	/Add registers 100 and
011	TAD 200	/200
012	SZA	/Is the result zero?
013	JMP 16	/No
014	TAD 100	/Yes, ADD register 100.
015	DCA 200	/Deposit and Clear Accumulator
016		/What is the accumulator here?- at the start of Concept j.

It is evident that the Accumulator may contain two different numbers when the program reaches register 16. If the comparison at register 12 produced a zero result, the instructions in registers 14 and 15 would be executed. The Accumulator would then contain zero upon reaching register 16. However, if the comparison yielded a non-zero result, register 13 would cause a direct jump to the end of the program segment. In this case the Accumulator would contain a number which may or may not be important to the program. Clearly, the system must have two different representations of the Accumulator status before the current instruction for register 16 can be determined.

To implement these ideas, two variables called ACC and ACCUM are used. ACC is a single variable which represents the state of the Accumulator assuming that the program sequence was executed sequentially with no jumps. ACCUM is a 35-element array representing the Accumulator contents at the start of each Concept Routine.

If ACC = i the Accumulator would currently be in state i were the program to follow a sequential path. The variable i can assume

one of the following values:

0 - representing an empty Accumulator,

1 - denoting valid numerical contents, or

2 - indicating a number which has no value to the current program.

Similarly, ACCUM indicates the Accumulator status at any specified point in the program. If ACCUM(j) = i then the Accumulator will be in state i when the program enters Concept Routine j.

In the example given above, the Accumulator contents would be zero at register 16 if the program was executed sequentially so ACC would therefore equal 0. However, the contents would be non-zero if this point in the program was reached by the jump in register 13 so ACCUM (j) would be set to either 1 or 2 depending upon the significance of these contents. The sequence of instructions beginning in register 16 would therefore have to reflect both possible states of the Accumulator in order to avoid simple but important errors in the final program results.

This Accumulator information is important to all Concept Routines because it determines whether certain program actions will be necessary such as clearing the Accumulator or retrieving a stored number for further processing. All Concept Routines expect the Accumulator to be in a particular state upon entry to the routine and if this is not the case, then instructions must be inserted in the program to bring it to this state. Both Accumulator variables are used to make this determination.

At the conclusion of any Concept Routine, the variables ACC and ACCUM are adjusted to reflect the program actions which occurred within the routine. This may involve setting the ACCUM array to

reflect conditions at a location which has not been reached yet. In our example, this would be done when the instruction in register 13 was inserted. The Concept Routine which was active at this time would note that the Accumulator would not be cleared when register 16 was reached and would therefore set the corresponding element of the ACCUM array to indicate this.

D. Program Jumps

Although the student designs his solution program sequentially, he must constantly be aware of logical branching within the program. It is often the case that a forward jump must be made to a yet unknown destination in the program. Clearly the Control Unit must maintain records of all such unresolved forward jumps. To accomplish this, all system components have access to an array variable, JMPLOC, defined as follows:

If JMPLOC(j) = k, there exists an unresolved jump in register k which should be resolved when concept j is entered. The Register k should then be filled in as a jump to the first register of concept j. To determine whether concept j has been reached yet, the Control Unit maintains another array, START, such that if

START(j) = m ,

then the segment of the student's program representing concept j began in memory register m.

To illustrate, consider the following program:

Assume that the Concept Sequence is '232416...'

This will give use to a program such as the following:

Memory Location	Number of the Concept which is represented by the program segment
10	
.	
.	Concept #23
.	
20	
21	
.	
.	Concept #24
.	
27	
30	
.	
.	Concept #16
.	
45	
46	

From this example we see that concept 23 starts in register 10, concept 24 in register 21, etc. The START array would therefore contain the following values:

START(23) = 10

START(24) = 21

START(16) = 30 ,

Now, if the instruction in register 46 should be a "JMP" to concept 10, it could not be coded at this time because concept 10 has not yet been reached in the program. Instead, the 10th element of the JMPLOC array would be set to 46 to indicate this condition. Later, when concept 10 was reached, this instruction could easily be inserted into the program.

These two arrays provide the system with the ability to interconnect various concepts much the same as a complex assembler or loader resulves external global references in a group of programs. It also is clearly illustrative of the fact that the Control Unit is keenly aware of all characteristics and parameters within the user's program.

The Control Unit guides the student through an orderly presentation of the material until the Concept Sequence, SEQ, has been reduced to a null indicator. At this time, the system must re-initialize many of its parameters and deal with any existing unresolved program jumps. If the entire program has not been written, the next portion of SEQ is used to generate a list of sub-tasks and the student continues to design his program. If, however, two contiguous null indicators are encountered in SEQ, the program is complete. The system will then generate a complete listing of the student's program for his future reference and begin its evaluation of the student's performance. Most of the student's records will be updated to include the current problem and he will be asked if he desires to continue operation. This process continues until the user indicates that he wishes to terminate operation in which case his performance on the entire series of problems is evaluated and entered into his permanent file.

It should be evident that while performing numerous clerical functions, the Control Unit also is responsible for a relatively complex analysis of the user's program. At any given point in the development of the program the Control Unit is aware of its exact nature and structure. As each concept is called and the program is expanded this knowledge must be constantly updated and revised to

reflect the immediate parameter conditions. It is for these reasons that the system Control Unit is a very general and useful CAI programming monitor.

Now that the operation of the overall system is clear, it is imperative that one consider how each respective programming concept is taught. Chapter five considers this topic and provides a discussion of the system's Concept Routines.

concepts such as table searching or input and manipulation of numbers require quite large programs to implement them. Smaller concepts such as Accumulator or Link manipulations can be taught quite simply.

Each Concept Routine is responsible for the design of a particular segment of the final user program. It therefore must have available to it all parameters in the program which are relevant to that segment. There is a kernel set of program parameters which are passed to every Concept Routine. This set includes the present memory configuration (MEM), the program counter (PC), the literal pointer (LIT), the concept sequence (SEQ), and the user's current LEVEL. Other parameters are available as they are needed in particular Concept Routines.

There are three distinct phases in the operation of every Concept Routine. Though these phases vary greatly in size and complexity, it is important that each be treated as a separate entity. The three portions of every Concept Routine are a generation phase, a teaching phase, and a remedial phase. Figure 5 indicates the basic flow of any Concept Routine.

possibilities. When the system finally decides that the response supplied by the student is in error, it informs him as to the reason for this decision and supplies the best program alternative.

If the student's response matches any of those which the system generated then it is accepted by the system as a valid program instruction. Since this was not the expected result however, several program parameters may have to be altered to adjust to this condition.

A trivial example of such a case occurs in the initialization of a counter. If a program loop is to be executed N times, then either the value of N or -N must be stored in memory. If the student chooses to use N as a counter value, he must form its 2's complement before he deposits it into the counter register. The system must make note of the method he chooses because if another counter is required later in the program, this same method of initialization must be used.

Despite the simplicity of this example, the principle in operation is important. Figure 7 gives a representation of the first method of program verification as used by the system.

Program Simulation

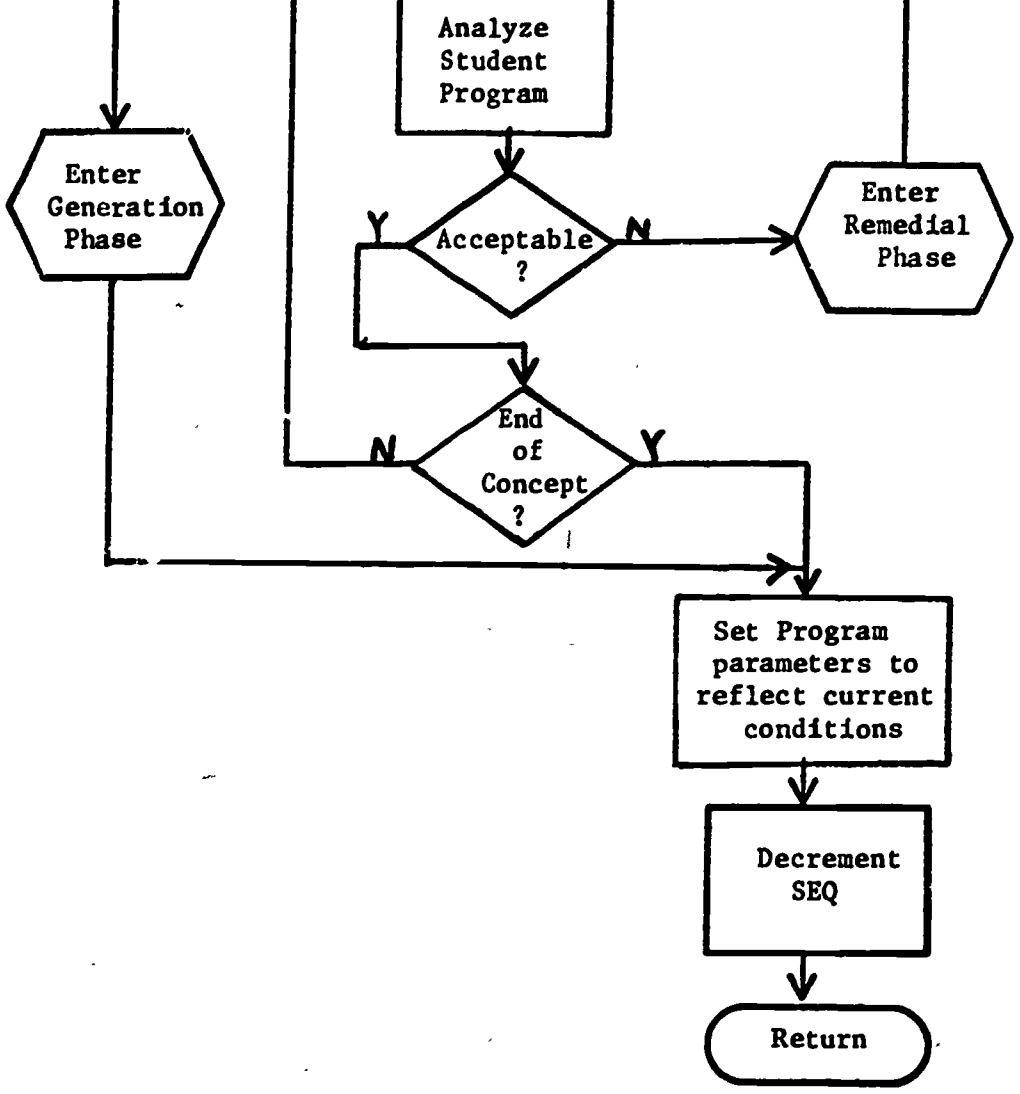
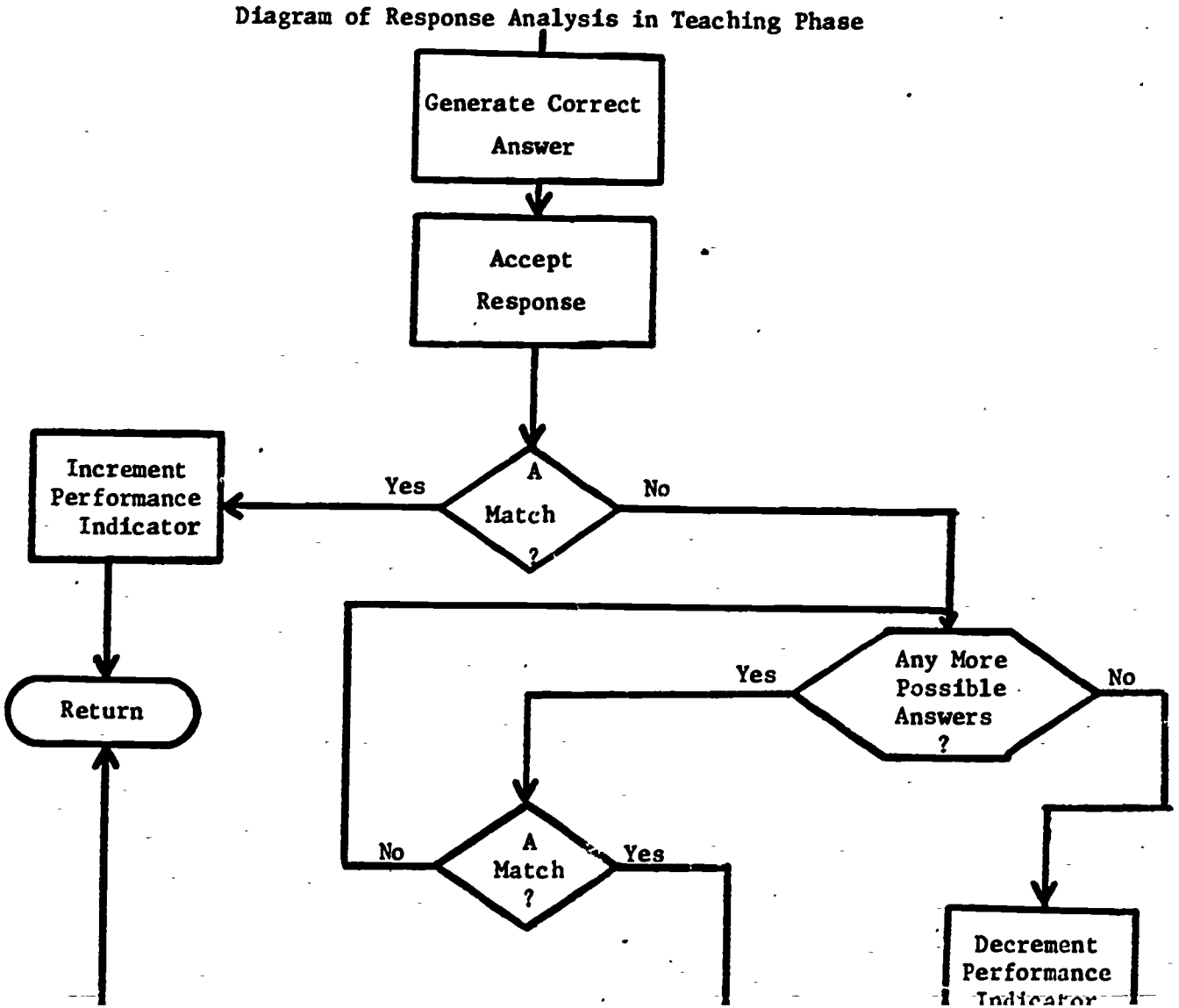


Figure 7

Diagram of Response Analysis in Teaching Phase



Generation Phase

The generation of a program segment means that the system completely designs and writes the program for the user. Since each Routine has this capability, the system's generative facilities are quite flexible.

The generation phase is entered only if certain conditions are present. The reasons for this are basic to the design of MALT. The system should obviously refrain from generating the entire solution program because the student would derive minimal benefit under these conditions. Likewise, the student should not have to write every portion of his program because many of the concepts involved may be quite trivial to him. The system therefore sets a particular generation threshold value for each Concept Routine. If the student's current LEVEL is above this value, the system generates a solution program segment for him. If his LEVEL falls below the threshold the system performs no generative function whatsoever and the student must design the entire program segment with only system assistance.

The numerical value of this threshold varies greatly with the difficulty of the concept involved. Quite easy topics such as transferring a number between the Accumulator and Memory may be system-generated at a LEVEL of 1. Others, such as table searching or list manipulation, are generated only at a LEVEL of 2.5. The result is that as a student progresses through the material, topics which he has previously mastered are performed by the system. He is responsible only for concepts which are commensurate with his current achievement level. Table 8 provides a sample of some Concept Routines

and their corresponding generation threshold. The third column of the table will be discussed later.

The generation threshold value of many Concept Routines can be extended in certain cases. A situation might arise in which a student has reached a particular level and has not yet been exposed to certain concepts which are usually encountered at lower levels. Although these concepts will probably prove fairly basic to him, he should nonetheless be given the chance to try them. Therefore, if a student's LEVEL places him into the generation phase of any concept which he has not yet encountered, he is often required to write the program for this concept himself. The system thereby virtually eliminates the possibility that a student may progress through the entire instructional sequence and never be exposed to certain basic programming concepts.

The generation phase of a Concept Routine does not consist of specific programs designed by the instructor. He provides only algorithms which the system can use to design its own programs. This is because user program parameters often vary and the programs which are generated must reflect these differences. The system must be able to design a correct program regardless of the initial contents of the Accumulator, the location of pointers or counters, etc. As the status of the user program changes, so too must the generation phase be able to alter its solution program to indicate these changes.

An important consideration in the automatic generation of a user's program is that of efficiency. Certainly the course instructor is capable to providing the system with complex algorithms which would produce programs that are maximally efficient. However, if these programs cannot be understood and followed by the average student,

Table 8.

SAMPLE CONCEPT ROUTINES,
GENERATION THRESHOLD, AND
INTERACTION RANGES

FUNCTION AND NAME OF CONCEPT ROUTINE	GENERATION THRESHOLD	VALUE OF STUDENT'S "LEVEL" WITHIN EACH INTERACTION RANGE
Use of pointers (PTR)	1.5	(0,.8),(.8,1),(1,1.5)
Use of counters (CTR)	1.5	(0,.8),(.8,1),(1,1.5)
"Masking" operations (mask)	1.5	(0,1),(1,1.5)
Textual output (TEXT)	1.7	(0,.5),(.5,1),(1,1.7)
Manipulation of LINK (oflow)	2.0	(0,.8),(.8,2)
Loop operations (lend)	2.0	(0,1),(1,2)
Output of numbers (preg)	2.5	(0,2),(2,2.5)
General input/output (read)	2.5	(0,1),(1,2.5)
Numerical comparisons (large)	2.5	(0,2),(2,2.5)

their instructional value is certainly lost. The system therefore must reach a reasonable trade-off between these two characteristics if it is to provide the student with the most possible benefits.

The present system appears to be quite effective in this respect. All programs generated by the system are as concise as can be expected. Still, the programs do not use any highly sophisticated techniques such as extensive microprogramming, self-modifying programs, or recursive subroutines. These methods would only serve to confuse a beginning student.

During the course of any program generation it may be necessary to call other Concept Routines to function as sub-concepts. A routine designed to teach simple textual output may want to use one which deals only with the operation of the printer, for example. Such concept interactions are common and cause no problems because if the highest level concept is being generated by the system, clearly any lower level ones would also be generated. These more basic concepts would usually be easy enough for the student that he would become quite bored if required to program them completely.

As each sub-task is generated by the system the resulting program is shown to the student. This allows him to monitor the construction of his program and note its logical flow.

In most Concept Routines, program generation is accomplished fairly quickly. It often appears to take longer than it actually does, however, due to the lack of dialogue. Nevertheless, the current wait time is about ten seconds, a tolerable interval when one considers the amount of system activity which occurs during this time.

The program generation facilities of the system have heretofore

proven to be successful. Some students tend to become impatient during generation but their boredom at being required to perform trivial programming tasks would probably be greater. The only difficulty encountered to date is that a problem is sometimes generated which is difficult in nature but can be broken down into a sequence of fairly basic sub-tasks. The student may then be faced with a situation in which the entire program is generated by the system even though his LEVEL is less than the maximum value of 2.5. This occurrence is obviously undesirable but can be remedied only by severely restricting the powers of the problem generator. It is therefore accepted as an unfortunate by-product of the system design.

Teaching Phase

All programming instruction which the student receives occurs during the teaching phase of a Concept Routine. This phase is entered whenever the student's LEVEL is less than the generation threshold value. This means that the concept will be difficult enough so as not to seem trivial to the student. In it, the student designs his own solution program under the guidance of the system.

Within the teaching phase of any Concept Routine there exists a variable number, usually two, of degrees of interaction. The student's LEVEL determines the amount of questioning and dialogue which the system initiates. At the lowest interaction range the system will guide the student through every step in the construction of his program. His program is required to be very similar to the system's solution program.

Figure 6.

EXAMPLE OF SYSTEM INTERACTION

AT VARIOUS "LEVELS"

0 < LEVEL < .8

```
Since we may need a ctr later, we will store the ctr value in
memory and move it into a register which we will use as the
actual ctr.
Use M377 to store the ctr value and M376 as the ctr.
You should use "7760" (the 2's complement of "0020") as the ctr value
M377:
7760
First, get the ctr value:
M001:
1377
And deposit it into the ctr:
M002:
3376
```

.8 < LEVEL < 1

```
Use either "7760" or "0020" as the ctr value.
Use M377 to store the ctr value and M376 as the ctr.
M377:
0020
First, get the ctr value:
M001:
1377
Now, form it's 2's complement:
M002:
7041
And deposit it into the ctr:
M003:
3376
```

1 < LEVEL < 1.5

```
Use M377 to store the ctr value and M376 as the ctr.
M377:
7760
M001:
1377
M002:
3376
```

user and tends to strengthen the possibility that they will occur again.

The MALT system was designed around the philosophy that program inefficiencies are to be allowed but always pointed out to the student. The reason for the inefficiency of the particular instruction involved is to be explained and a better alternative provided. However, errors which disrupt the logical flow of the program are never to be allowed. To do so would make the final results totally invalid and would teach the student virtually nothing about correct programming. As each logical error in the program is typed by the student, the system explains why it is inappropriate to the current process. It also gives the current instruction and usually inserts it into the program.

There are certain instances, however, in which the correct instruction is not inserted into the program by the system. Early results from the student's use of the system seem to indicate that the primary cause of programming error involves use of the Accumulator. Students tend to disregard the current status of the Accumulator when formulating their program. The obvious result is that this register often contains a number when it should be cleared or vice versa. Since this type of error is so important to the final results, the system requires the student to correct the program himself. For example, if a student neglects to issue an "CLA" instruction (CLear the Accumulator) when one is absolutely necessary, the system requires him to do so regardless of his current LEVEL. This technique forces him to focus his attention on the instruction and its use in the current program. It also prevents a common mistake from being overlooked

and developing into an undesirable programming habit.

During the teaching of any large concept it is often necessary, as it was in the generation phase, to use other smaller sub-concepts. There is a marked difference in the treatment of these sub-concepts, however, because of a sub-concept called from the teaching phase of a Concept Routine may or may not enter its own generation phase. That is, even though the sub-concept is obviously more basic than the concept which called it, it may still be difficult enough so that the student is required to program it. The system can thereby adapt very closely to the student's needs because even though he may be faced with a very difficult programming concept, various portions of it can be generated for him by the system. He is free to focus his attention on the more challenging segments of his program.

Every response which the student makes during the teaching phase is ultimately reflected in his LEVEL. He is usually penalized a significant amount for errors in a particular concept while being rewarded a lesser amount for correct responses in the same concept. The result is that the student's LEVEL tends to converge quickly to a value which very nearly reflects his current programming ability.

Whenever an incorrect response is typed by the student, the system temporarily exits from the teaching phase of the active Concept Routine. It enters the remedial phase in which the student is provided with an explanation as to the reason for his error and a possible correct solution. The system will subsequently re-enter the teaching phase to finish guiding the student through his solution program.

Remedial Phase

The final portion of any Concept Routine has the responsibility of generating remedial statements to indicate to the student the reason that his program is in error. These remedials usually include an explanation of the error and a description of the steps necessary to correct it.

The remedials given by the system are rarely pre-determined by the instructor. They are usually simple sentence formats from which an appropriate response can be constructed. A wide variety of possible student errors are therefore handled effectively by only a few statement formats with no loss of remedial ability.

The system usually provides the correct program statement in both machine and assembly language formats. The conversion from one format to the other is trivial for the system and shows the student whether his error was a logical or merely a syntactic one. It also helps him to develop the habit of formulating his solution program first in assembly-language which tends to minimize purely syntactic errors.

Remedial statements of more than one sentence are sometimes generated semi-randomly according to a particular function. The same user mistake will often result in the generation of two different remedials by the system. The student may therefore possibly receive either a basic or a complete explanation as to the cause of his error. The function which determines this is heavily weighted so that complete remedials are more likely to occur if the student's LEVEL is relatively low than if it is high. This helps to insure not only that errors are fully explained and understood by the student, but also that remedials are consistent with his present ability.

The three phases of a Concept Routine are designed to maximize the flexibility of the system. Each phase has a particular function which varies somewhat according to the student's past performance. The inclusion of three such phases into every Concept Routine allows the system to perform effectively for all students regardless of their abilities. Appendix B illustrates the use of the teaching and remedial phases during the programming of a simple concept.

The system also has the ability to simulate execution of the user's program if it feels this process will be beneficial to the student. Chapter 6 discusses the method by which this is accomplished.

VI. PROGRAM VERIFICATION AND SIMULATION

There are two important techniques used by the MALT system to judge the correctness of a student's program. The most common method is to analyze in detail each segment of the program as it is typed in to determine if it performs the required functions. This is done on an instruction-by-instruction basis so that there is immediate feedback to the student. In the rare cases where this method is not feasible, the program can be simulated and the results of this simulation analyzed by the system. This method involves much system activity and will be considered in detail. In either case, the system has virtually total ability to recognize logical errors in the user's program.

Immediate Program Verification

The first technique of program verification requires that the system be thoroughly familiar with virtually all aspects of the current program. Only if it is aware of the status of all program parameters can it effectively judge the merits of any program segment which the student might design. It also must be able to determine not only if the response is the best possible one, but also if there exists other alternative responses which also are acceptable.

As the student formulates each response, the system also generates what it considers to be an appropriate answer. If the two do not match, the system must determine if other responses are possible. If so, the student's answer is compared with all such reasonable

many possible solutions, each as valid as the next, that to force the student to use any particular one would deprive him of much of his programming freedom. It would also clearly be counter-productive and contradictory to the original aims of this research.

It was decided, therefore, that program execution used sparingly was the best alternative. This method sharply decreases the amount of correcting ability in the system but provides the student with greater freedom in his program construction. The trade-off is very worthwhile in the cases in which it is used because to provide the system with full program monitoring abilities would be prohibitive in terms of time and disk storage. Since the system is concerned only with the final results of the program and not each intermediate step, the student is free to design his program using virtually any method he feels is effective.

The limited execution of a user's program provides other advantages also. Most students learn basic programming by turning in a set of cards at their computer center and returning later for the output. The intermediate results and actions of the program are unknown to them. By actually observing their program in execution they gain insight into both its logical structure and the nature of any errors which might be present. There are only 5 concepts for which simulation is used. These include the following:

1. Manipulation of the Link register
2. Comparison of register contents
3. Movement of numbers within memory
4. Decisions based upon the Accumulator sign bit
5. Decisions based upon the absolute magnitude of the Accumulator

Despite the apparent simularities between the MALT system and previous ones in this respect, there is a fundamental difference. In MALT, the student is not expected to judge the correctness of his own program. The system analyzes the program both before and after simulation in order to isolate any possible errors. This procedure guarantees that the final program will be totally correct in its logical structure.

The actual operation of program verification is very similar to the process a student follows when he de-bugs his program on a step-by-step basis. The primary difference is that the system makes evaluative decisions concerning the program while this responsibility is usually left up to the programmer. That portion of the system which makes these decisions is called the Program Verifier.

Before any simulation is attempted, the current status of the user's program must be saved. This is important because it eliminates the possibility that the execution of a new program segment may modify or destroy other portions of the program. Next, the operation which the program segment is supposed to perform is determined by a numerical code which is passed to the Verifier. Each such code is unique and determines the proper state of the machine before and after execution.

After the intent of the program segment is established, all conditions of the machine which might possibly affect final program results are determined. This is absolutely essential because the user's program must be tested under all possible machine states. To do otherwise would make the system's judgements pure guess-work.

These initial conditions set by the system are determined by the characteristics of the program segment itself. For example, if the

program is intended to perform a particular operation depending upon the status of the Link register, then only two initial states are necessary. The program is tested with a zero Link and again later with a non-zero one. Since the flow of the program depends solely on these two conditions, to test it under other initial states would be superfluous.

The duties of the Verifier are temporarily suspended upon determining each initial machine condition. The SEDCOM Simulator is then called in order to execute the program. This simulation is also subject to certain constraints and will be discussed below.

When the Verifier regains control of the system, several determinations must be made. It is quite possible that program execution was not terminated normally. Abnormal conditions which would cause simulation to cease are such things as infinite loops, undefined instructions, or incorrect program branches. These conditions must be corrected immediately before any other analysis of the program is initiated. Therefore, the student is given the opportunity to correct his program, the current set of initial conditions are re-established, and simulation is attempted again.

If normal program termination occurred, the system must proceed to analyze the results to determine their validity. This is accomplished by implementing the concepts derived in Manna's research [7]. For each initial machine state, $\phi(\bar{x})$, the user's program must perform a particular function, $z = f(\bar{x})$. If this function is performed correctly, the corresponding output predicate, $\psi(\bar{x},z)$ will be true. If this is the case, the program is judged to be correct for the given initial states.

The determination of the appropriate output predicate is not trivial. Many terminal conditions must be checked such as the contents of any memory register, the status of the Accumulator and Link and the location where the program halted execution. Each of these parameters must also be analyzed in relation to other such parameters. For example, it may not be enough that a given register holds a particular value. This value might have to be compared in some way with the contents of another register involved in the program. A flow chart illustrating operation of the Program Verifier is given in Figure 8.

If any particular terminal condition indicates that the user's program did not perform its function correctly, the system attempts remedial action. Since it is aware of the exact results which should have been obtained, it can provide a concise description of the error. It cannot however, by its very nature, isolate the location of the error in the user's program. This determination is left up to the student. However, the problem has been greatly simplified due to the system's diagnostics and the user's ability to observe his program in execution. An example of actual Program Verification operation is given in Figure 9. The sample program finds the absolute value of the ACC.

The Program Verifier will not allow any program to be judged correct if it does not meet stringent standards. This implies that each program must be tested and modified until it is acceptable to the system. The time involved in such a process is usually small but if a student continues to have difficulty it may become quite large. For this reason, the student may press the ATTN button at any time to halt program simulation. Besides the usual alternatives given to him, he is offered a special option merely to return to the latest concept

Figure 8

Flow Chart of Program Verifier

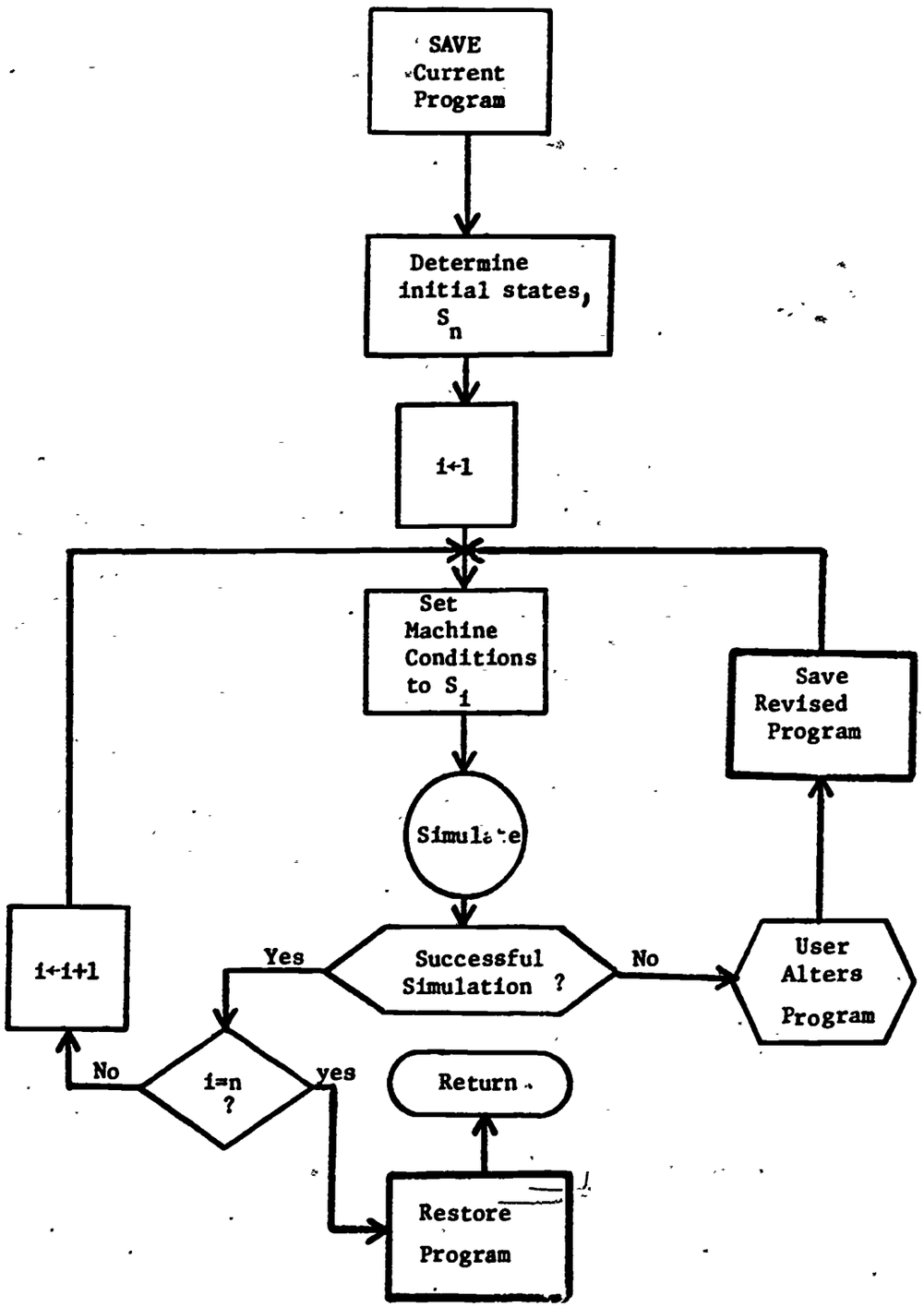


Figure 9.

OPERATION OF PROGRAM VERIFIER

```
Type in your program for this sub-task and end it with "OK".
001 :
_7500
002 :
_7402
003 :
_7040
004 :
_7402
005 :
_OK
I can't tell if this is correct. Should I execute it?
_yes

Watch as your program is executed:
First, let the (ACC) be negative.
001 : 7500
003 : 7040
004 : 7402
Your program has halted because of a "7402" command.
Is this what it was supposed to do?
_yes
Your program appears to have an error in it!
if the ACC is negative, you are not forming the 2's comp. of it.
Do you need to change your entire program segment?
_no
How many registers do you need to change?
number
_2
Type in the address, hit carriage return, then the instruction.
_004
_7001
_005
_7402
First, let the (ACC) be negative.
001 : 7500
003 : 7040
004 : 7001
005 : 7402

Now, let the (ACC) be positive.
001 : 7500
002 : 7402
Your program has halted because of a "7402" command.
Is this what it was supposed to do?
_yes
Congratulations!!!  I think your program segment works.
```

entered instead of terminating system operation entirely.

The ultimate acceptance of the program by the system is based upon its correct simulation under each initial condition. Once this is done, the original or corrected copy of the program is restored. This is necessary again because even successful program operation may have altered the contents of certain registers which should remain invariant.

SEDCOM Simulator

The simulation process in MALT is made far more complex than it need be simply due to the structure of the CPS system. The Simulator would be almost a trivial addition if CPS were designed similarly to the structure of a small computer. This is not the case however, so simple program interpretation becomes an involved affair. All manipulations, including arithmetic operations, must be performed on character strings representing octal digits. The result is that much system programming effort is expended on enabling the system to perform relatively trivial tasks.

That portion of the system which actually performs the simulation is fairly straightforward, though somewhat bulky. Since the CPS language is the vehicle for this instructional system, the SEDCOM Simulator must obviously function in an interpretive mode. The Verifier provides it only with the starting and correct ending location of the program segment written by the student.

The Simulator is designed to operate much the same as the actual PDP-8 computer. Each hardware register in the PDP-8 has a software counterpart in the Simulator. The primary registers used by the Simulator are a Memory Buffer Register (MBR), Memory Address Register

(MAR), Defer Register (D), Instruction Register (IR), and Current Instruction Counter (CIC). Each of the registers is stored as a fixed-length character string. The MAR and MBR are used to contain, respectively, the address of any memory registers which is referenced and the contents of that register. The D register contains the status of the address mode bit which indicates either direct or indirect addressing. The CIC indicates the next instruction in the program to be executed. Finally, the IR contains the operation code of the current instruction.

There are two types of instructions with which the Simulator must deal. These are memory Reference Instructions (MRI) and Register Reference Instructions (RRI). The first type involves operations on memory registers while the latter deals only with manipulation and testing of the Accumulator and Link contents. These groups of instructions are handled in two completely different ways by the system.

To simulate execution of an MRI, several steps must be performed in the proper order. First, the contents of the CIC are loaded into the MAR and the contents of the register addressed by the MAR is put into the MBR. Next, the operation code of the instruction is extracted and placed in the IR. The address mode bit is loaded into the D register. The address field of the instruction can then be loaded into the MAR and the contents of the memory register referenced by this address is placed into the MBR. If indirect addressing is indicated, this last process is repeated using the MBR contents as an address field to get the proper number into the MBR. The instruction is then simulated using the MBR contents as the operand and the

operation code to indicate the correct operation. In general, the CIC is incremented by one after an MRI and sequential simulation continues. There are instances, (a JMP or ISZ instruction), where the contents of the CIC may be considerably altered after the simulation of a particular instruction.

Register Reference Instructions cannot be simulated using the same methods as the MRI. Each bit of these instructions corresponds to a particular operation to be performed. If multiple bits are set to "1", then multiple operations are necessary on the Accumulator or Link contents. Therefore, each bit is analyzed in turn and the corresponding operation is either performed or ignored depending on the contents of the bit.

The process of program simulation continues until the end of the program segment is reached. This indicates normal program termination and control is then returned to the Verifier in order to test the results of the simulation.

As mentioned above, there are several errors that may exist in a program which are independent of the purpose of that program. These mistakes must be corrected immediately even before any other determination of program correctness is attempted. The Simulator therefore functions not merely as a vehicle for program execution, it also has the capacity to make limited judgements about the basic structure of the user's program. These determinations are strictly syntactic however, and are not to be confused with the subjective, semantic corrections which the Program Verifier makes.

One of the simplest checks which the Simulator makes is that of locating undefined instructions. The student is shown each such

instruction and immediately required to correct it. A similar situtation occurs when an instruction is found in a program in which it clearly does not belong. Since only a limited number of program segments are candidates for simulation, there are some instructions which would be incorrect if included in these segments. For example, no program segment involving input/output programming is ever simulated. Therefore, any I/O instruction which is found by the Simulator is definitely out of place and must be changed.

The Simulator also has a few more subtle monitoring abilities. All user programs are contained to operate within the range of the current program segment. Any attempt to branch out of this area or to make unauthorized register references will be suppressed by the Simulator. Also, the program is constantly monitored for the existence of infinite loops. If such a loop appears to exist, simulation ceases and corrective measures are taken by the student. Appendix C illustrates a case in which an unauthorized branch was made in the student's program.

As any error is uncovered by either the Simulator or the Verifier, the student is asked to make the necessary revisions in his program. Although this process often requires a redesign of the entire segment, it sometimes calls for only the correction of a few instructions. To avoid re-typing of those parts of the program which have already been accepted, the system allows the student to correct only a few instructions if he so desires.

The verification and simulation properties of MALT were introduced as a method of enhancing the flexibility of the system. Students tend to tire of constant dialogue and become bored unless the system

progresses them through the material with reasonable swiftness. By observing his program in action, the student is spared long waits and needless interaction with the system. It also greatly simplifies those Concept Routines in which it is used yet causes only a minimum loss of program correction ability. These basic reasons, along with the fact that student reation to this facility has been very favorable, seem to justify its inclusion in the MALT system.

It should now be clear exactly how the system attempts to teach machine-language programming. The final matter to consider is the way in which student performance records are maintained. Chapter 7 provides a discussion of the CPS file system and how these files are stored and evaluated by the system.

VII. STUDENT RECORDS

In order for the system to adapt its presentation to each student, records of all student performance must be maintained. These records are kept in three CPS files stored on a random access disk for quick retrieval.

CPS has facilities for records to be stored in disk areas other than those reserved for executable programs. This allows larger CPS programs to be used because the programmer need not concern himself with reserving storage for a large number of records. These files can be retrieved and updated selectively so that a minimum amount of access time is consumed. The current time spent in file retrieval is only about four or five seconds. This delay is completely tolerable especially because it only occurs after the student has finished his solution program.

MALT maintains student records in three separate CPS files. Two of these files are reserved for records of each individual student and the third is used as a cumulative class performance indicator. At the beginning of any semester, the instructor can reset all of these files to contain null elements. This can be accomplished completely by the system using facilities in the Instructor Mode.

As each student uses the system for the first time, a fresh file record is established for him. The contents of the file are initialized to reflect that the student is a beginning user. After each system use, the student's records are updated to indicate his latest achievement level.

The cumulative record of each student is kept in the file SNAMES. A diagram of the structure of this file is given in Figure 10. This is a sequential file and occupies 352 bytes of disk storage. It contains sixteen elements, each one being a character string of length twenty-two.

The first eight characters in a SNAMES file entry is the student's name. Columns nine and ten indicate the user's LEVEL value after his last use of the system. This number is always rounded off to the nearest tenth after the student finishes a session at the terminal. The next two columns reflect the number of problems which the student has solved. This enables the instructor to judge the relative progress of each student because his can compare the change in LEVEL (from the initial value of .3) to the number of problems needed to bring about this change.

Columns thirteen through fifteen of the SNAMES file show the total time in minutes each student spent using the system. Certain deductions concerning response time can be made from this parameter although each such conclusion would be highly speculative. The next four columns indicate the total number of responses which the student made. Columns twenty through twenty-two show how many of these responses were incorrect. These two parameters can be used to determine an actual numerical grade to adequately reflect the student's complete performance.

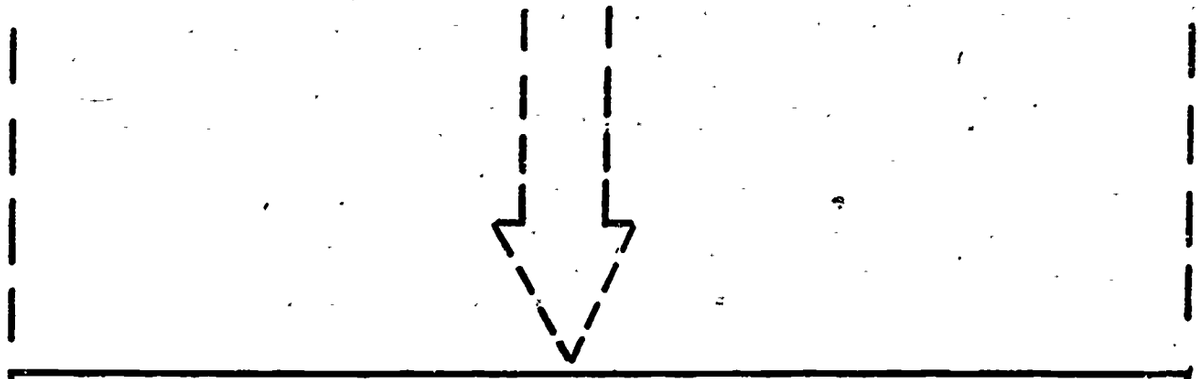
There is a fundamental reason why the system maintains records of incorrect student responses as opposed to correct ones. In many subjects taught through the use of CAI, every response can easily be classified as being either right or wrong. In computer programming

Figure 10.

STUDENT FILE SHAPES

SAMPLE ENTRIES

COLUMNS 1-8	9-10	11-12	13-15	16-19	20-22
#1 STUDENT NAME	LAST LEVEL	USES	TIME	RESPONSES	ERRORS
#2 STUDENT NAME	1.6	08	047	0090	018
#3 STUDENT NAME	0.5	02	013	0017	004
#4 STUDENT NAME	2.1	15	120	0110	014



#16 STUDENT NAME	1.1	05	022	0040	003

however, this is not the case. A particular sequence of machine-language instructions which the student types in might not be exactly the response expected by the system. It may be, however, perfectly acceptable in that it performs the correct function with only a small loss of program efficiency. Such a situation should be noted to the student but the program segment should not be considered as being incorrect. Only sequences which obviously will not bring out the desired results should be treated as incorrect. The system maintains records, therefore, only of student responses which clearly were inappropriate to the process involved.

The SNAMES file is initialized only before the start of a semester. As each new student enters the system a record is allocated to him and his name entered into it. All other values in the file are set to zero, except his LEVEL which is initialized to 0.3. This file is updated only upon completion of an entire session at the terminal. After this occurs, the system logs out the terminal to avoid a student gaining unauthorized access to his file.

The complete history of each student's use of the system is provided by the PROBS file, shown in Figure 11. It is regional in organization and occupies two disk tracks. Because of its organization and size, only one student's record is accessed or updated at a time.

There is currently space for the records of twenty problems for each student. The results of each use of the system by the student is represented as a seven-element array. Since the PROBS file can be reset by the instructor at any time, there are no difficulties concerning space limitations.

Figure 11.

STUDENT FILE PROBS

16 ENTRIES

SAMPLE ENTRY FOR THE j^{th} STUDENT

	ENTRY	VALUE
PROBLEM #1	1	1st PROBLEM PRIMITIVE NUMBER
	2	2nd PROBLEM PRIMITIVE NUMBER
	3	3rd PROBLEM PRIMITIVE NUMBER
	4	LEVEL ON THIS PROBLEM
	5	TIME (IN MINUTES)
	6	RESPONSES
	7	ERRORS
	⋮	⋮
PROBLEM #20	1	5
	2	9
	3	2
	4	2.1
	5	17
	6	26
	7	2

The seven elements of each entry are designed to provide all the information which the system requires in evaluating student performance on a particular problem. The first three entries indicate the number of each respective primitive which were used in the problem. This enables system alterations to be made if problem generation is not sufficiently random. The fourth element is the student's final LEVEL after solving this problem. Entries five through seven are equivalent to their counterparts in the SNAMES file. They represent, respectively, the time spent on this problem, the number of responses made, and the number of incorrect responses. These entries enable the instructor to isolate certain types of problems which are proving difficult to a particular student.

The third system file, RATIOS, focuses on general class performance, not individual achievement. This file is also direct-access regional but requires only one track. It consists of a 2 x 13 numerical array with the second subscript in the range (0:12).

Each of the twelve principal elements (the entry with the '0' subscript is a dummy record) represents the class performance on a particular programming concept. Each such element is termed a Concept Ratio, (see Table 9). These concepts are more general than those of the Concept Routines because a broad achievement indicator which has some diagnostic validity was desired. Several related Concept Routines may be combined to form a larger Concept Ratio. For instance, the second Concept Ratio is entitled "Textual Output" and represents a host of programming operations. By including all these minor operations into one large class, the instructor is able to achieve a better view of general class performance and to gear his

Table 9.

LIST OF CONCEPT RATIOS

CONCEPT RATIO	SUBSCRIPT OF RATIOS FILE
REG-TO-REG MOVEMENTS	1
TEXTUAL OUTPUT	2
READ/WRITE OPERATIONS	3
INPUT OF NUMBERS	4
TABLE SEARCHING OPERATIONS	5
ACCUMULATOR MANIPULATIONS	6
INITIALIZATION OF POINTERS	7
INITIALIZATION OF COUNTERS	8
OUTPUT OF REGISTER CONTENTS	9
SORTING TECHNIQUES	10
OPERATIONS INVOLVING PROGRAM LOOPS	11
MISCELLANEOUS REGISTER OPERATIONS	12

classroom presentation to the student's needs.

The technique of maintaining records only of broad areas of programming was chosen for another important reason. If quite detailed and specific concepts were utilized, small deficiencies in the system could produce profound effects. If a particular question or statement happened to be badly worded, it might cause virtually every student to respond incorrectly. The corresponding Concept Ratio score would be extremely poor despite the fact that the system was basically at fault. However, by enlarging the scope of each Concept Ratio to include many related, smaller operations, the effects of such an occurrence are completely minimized.

Each Concept Ratio has two components which determine its value. The first is the number of questions asked on a programming concept and the second is a total count of incorrect responses. These are stored in the Ratios array with the first subscripts 1 and 2 respectively. The value of the Concept Ratio for the j^{th} programming concept is:

$$\text{Concept Ratio} = 100 - \left(\frac{\text{Ratios } (2,j)}{\text{Ratios } (1,j)} * 100 \right)$$

The result is a percentage value indicating the extent to which the class mastered the corresponding programming concept.

The student records were implemented as a means of providing structure to the progression of a student through the system. They appear to perform this function well although it is basically only a clerical one.

This chapter concludes the discussion of the operation of MALT.